Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1

Document Number: 248966-050US
April 2024

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

REVISION HISTORY

Date	Revision	Description
January 2023	046	• Introduction to the 4th Generation Intel® Xeon Scalable family of processors. • Optimization of scalability and communications for the 4th Generation Intel® Xeon Scalable Family of Processors. • Intel® Advanced Vector Extensions 512 - FP16 Instruction Set for Intel® Xeon® Processors (Chapter 19). • Intel® Advanced Matrix Extensions (Intel® AMX) (Chapter 20). • Intel® QuickAssist Technology (QAT) (Chapter 22). • Update to Appendix B.
January 2023	046A	• Correction of author. • Correction of incorrect link.
May 2023	047	• Chapter 1: Introduction. • Chapter 2: Intel® 64 and IA-32 Processor Architectures. • Chapter 3: General Optimization Guidelines. • Chapter 7: Optimizing for SIMD Floating-point Applications. • Chapter 10: Sub-NUMA Clustering. • Chapter 11: Multicore and Hyper-Threading Technology. • Chapter 15: Optimizations for Intel® AVX, FMA, and AVX2. • Chapter 18: Software Optimization for Intel • AVX-512 Instructions. • Chapter 20: Intel® Advanced Matrix Extensions (Intel® AMX).
August 2023	48	• Chapter 2: Intel® 64 and IA-32 Processor Architectures. • Chapter 3: General Optimization Guidelines. • Chapter 5: Coding for SIMD Architectures • Chapter 7: Optimizing for SIMD Floating-Point Applications. • Chapter 23: Knight's Landing (moved to external document). • Created a Volume 2, including Appendix D, Appendix E, and Appendix F.
January 2024	49	• Accessibility updates: tables. • Chapter 1: Organized content into tables. added references to the latest microarchitecture. • Chapter 20: Added four columns to Table 20-5. and information related to the latest microarchitecture.
April 2024	50	• Organizational changes • Chapter 1: Added glossary and mentions of Meteor Lake Microarchitecture. • Chapter 2: Added Redwood Cove, Intel® 64 and IA-32 Instruction Best Practices, and Non-Privileged Instruction Serialization. • Chapter 4: Gracemont content updated to Crestmont microarchitecture; updated other content for accessibility. • Chapter 16: Moved to Volume 2. • Chapter 17: Updated Table 17-1, and added figure names.

CONTENTS

PAGE

CHAPTER 1
INTRODUCTION

1.1 ABOUT THIS MANUAL . . . 1-1
1.2 TUNING YOUR APPLICATION . . . 1-1
1.3 INTEL PROCESSORS SUPPORTING THE INTEL® 64 ARCHITECTURE . . . 1-2
1.4 ORGANIZATION OF THIS MANUAL . . . 1-4
1.4.1 Chapter Summaries . . . 1-4
1.4.1.1 Volume 1 . . . 1-4
1.4.1.2 Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures . . . 1-5
1.5 GLOSSARY . . . 1-6
1.6 RELATED INFORMATION . . . 1-7

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

2.1 6TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY . . . 2-1
2.1.1 The Redwood Cove Microarchitecture . . . 2-1
2.1.1.1 Branch Hint . . . 2-2
2.1.1.2 Profiling Support for Branch Hints . . . 2-3
2.1.1.3 New Redwood Cove Macro-Fusions . . . 2-3
2.1.1.4 Improved Memory BW . . . 2-4
2.1.1.5 Array of Pointers Prefetcher . . . 2-4
2.1.1.6 LLC Page Prefetcher (LLCPP) . . . 2-5
2.2 THE SAPPHIRE RAPIDS MICROARCHITECTURE . . . 2-5
2.2.1 4th Generation Intel® Xeon® Scalable Family of Processors . . . 2-6
2.3 THE ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE . . . 2-6
2.3.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture . . . 2-6
2.3.2 Hybrid Scheduling . . . 2-6
2.3.2.1 Intel® Thread Director . . . 2-6
2.3.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture . . . 2-9
2.3.2.3 Scheduling with a Multi-E-Core Module . . . 2-10
2.3.2.4 Scheduling Background Threads on x86 Hybrid Architecture . . . 2-10
2.3.3 Recommendations for Application Developers . . . 2-10
2.4 THE GOLDEN COVE MICROARCHITECTURE . . . 2-10
2.4.1 Golden Cove Microarchitecture Overview . . . 2-11
2.4.1.1 Cache Subsystem and Memory Subsystem . . . 2-15
2.4.1.2 Avoiding Destination False Dependency . . . 2-15
2.5 ICE LAKE CLIENT MICROARCHITECTURE . . . 2-16
2.5.1 Ice Lake Client Microarchitecture Overview . . . 2-16
2.5.1.1 The Front End . . . 2-18
2.5.1.2 The Out of Order and Execution Engines . . . 2-18
2.5.1.3 Cache and Memory Subsystem . . . 2-20
2.5.1.4 Fast Store Forwarding Prediction (FSFP) . . . 2-23
2.5.1.5 New Instructions . . . 2-24
2.5.1.6 Ice Lake Client Microarchitecture Power Management . . . 2-25
2.6 SKYLAKE SERVER MICROARCHITECTURE . . . 2-25
2.6.1 Skylake Server Microarchitecture Cache . . . 2-27
2.6.1.1 Larger Mid-Level Cache . . . 2-27
2.6.1.2 Non-Inclusive Last Level Cache . . . 2-27
2.6.1.3 Skylake Server Microarchitecture Cache Recommendations . . . 2-27
2.6.2 Non-Temporal Stores on Skylake Server Microarchitecture . . . 2-29
2.6.3 Skylake Server Power Management . . . 2-29
2.7 SKYLAKE CLIENT MICROARCHITECTURE . . . 2-33
2.7.1 The Front End . . . 2-34
2.7.2 The Out-of-Order Execution Engine . . . 2-34
2.7.3 Cache and Memory Subsystem . . . 2-36
2.7.4 Pause Latency in Skylake Client Microarchitecture . . . 2-37

CONTENTS

PAGE

2.8 INTEL® 64 AND IA-32 INSTRUCTION BEST PRACTICES 2-39
2.8.1 Non-Privileged Instruction Serialization 2-39
2.8.2 Intel® Hyper-Threading Technology (Intel® HT Technology) 2-40
2.8.2.1 Processor Resources and Intel® HT Technology 2-41
2.8.2.2 Microarchitecture Pipeline and Intel® HT Technology 2-42
2.8.2.3 Execution Core 2-42
2.8.2.4 Retirement 2-42
2.8.3 SIMD Technology 2-42
2.8.4 Summary of SIMD Technologies and Application Level Extensions 2-44
2.8.4.1 MMX™ Technology 2-45
2.8.4.2 Streaming SIMD Extensions 2-45
2.8.4.3 Streaming SIMD Extensions 2 2-45
2.8.4.4 Streaming SIMD Extensions 3 2-45
2.8.4.5 Supplemental Streaming SIMD Extensions 3 2-45
2.8.5 SSE4.1 2-46
2.8.5.1 SSE4.2 2-46
2.8.5.2 AESNI and PCLMULQDQ 2-46
2.8.5.3 Intel® Advanced Vector Extensions (Intel® AVX) 2-47
2.8.5.4 Half-Precision Floating-Point Conversion (F16C) 2-47
2.8.5.5 RDRAND 2-47
2.8.5.6 Fused-Multiply-ADD (FMA) Extensions 2-47
2.8.5.7 Intel® Advanced Vector Extensions 2 (Intel® AVX2) 2-48
2.8.5.8 General-Purpose Bit-Processing Instructions 2-48
2.8.5.9 RDSEED 2-48
2.8.5.10 ADCX and ADOX Instructions 2-48

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

3.1 PERFORMANCE TOOLS 3-1
3.1.1 Intel® C++ and Fortran Compilers 3-1
3.1.2 General Compiler Recommendations 3-2
3.1.3 VTune™ Performance Analyzer 3-2
3.2 PROCESSOR PERSPECTIVES 3-2
3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy 3-2
3.2.2 Transparent Cache-Parameter Strategy 3-3
3.2.3 Threading Strategy and Hardware Multithreading Support 3-3
3.3 CODING RULES, SUGGESTIONS, AND TUNING HINTS 3-3
3.4 OPTIMIZING THE FRONT END 3-4
3.4.1 Branch Prediction Optimization 3-4
3.4.1.1 Eliminating Branches 3-4
3.4.1.2 Static Prediction 3-6
3.4.1.3 Inlining, Calls, and Returns 3-7
3.4.1.4 Code Alignment 3-8
3.4.1.5 Branch Type Selection 3-8
3.4.1.6 Loop Unrolling 3-10
3.4.2 Fetch and Decode Optimization 3-11
3.4.2.1 Optimizing for Microfusion 3-11
3.4.2.2 Optimizing for Macrofusion 3-12
3.4.2.3 Length-Changing Prefixes (LCP) 3-16
3.4.2.4 Optimizing the Loop Stream Detector (LSD) 3-18
3.4.2.5 Optimization for Decoded ICache 3-18
3.4.2.6 Other Decoding Guidelines 3-20
3.5 OPTIMIZING THE EXECUTION CORE 3-20
3.5.1 Instruction Selection 3-20
3.5.1.1 Integer Divide 3-21
3.5.1.2 Using LEA 3-21
3.5.1.3 ADC and SBB in Sandy Bridge Microarchitecture 3-23

CONTENTS

PAGE

3.5.1.4 Bitwise Rotation . . . 3-24
3.5.1.5 Variable Bit Count Rotation and Shift . . . 3-24
3.5.1.6 Address Calculations . . . 3-24
3.5.1.7 Clearing Registers and Dependency Breaking Idioms . . . 3-25
3.5.1.8 Compares . . . 3-27
3.5.1.9 Using NOPs . . . 3-27
3.5.1.10 Mixing SIMD Data Types . . . 3-28
3.5.1.11 Spill Scheduling . . . 3-28
3.5.1.12 Zero-Latency MOV Instructions . . . 3-29
3.5.2 Avoiding Stalls in Execution Core . . . 3-31
3.5.2.1 Writeback Bus Conflicts . . . 3-31
3.5.2.2 Bypass Between Execution Domains . . . 3-31
3.5.2.3 Partial Register Stalls . . . 3-32
3.5.2.4 Partial XMM Register Stalls . . . 3-32
3.5.2.5 Partial Flag Register Stalls . . . 3-33
3.5.2.6 Floating-Point/SIMD Operands . . . 3-34
3.5.3 Vectorization . . . 3-35
3.5.4 Optimization of Partially Vectorizable Code . . . 3-35
3.5.4.1 Alternate Packing Techniques . . . 3-37
3.5.4.2 Simplifying Result Passing . . . 3-37
3.5.4.3 Stack Optimization . . . 3-38
3.5.4.4 Tuning Considerations . . . 3-39
3.6 OPTIMIZING MEMORY ACCESSES . . . 3-40
3.6.1 Load and Store Execution Bandwidth . . . 3-41
3.6.1.1 Making Use of Load Bandwidth in Sandy Bridge Microarchitecture . . . 3-41
3.6.1.2 L1D Cache Latency in Sandy Bridge Microarchitecture . . . 3-42
3.6.1.3 Handling L1D Cache Bank Conflict . . . 3-43
3.6.2 Minimize Register Spills . . . 3-44
3.6.3 Enhance Speculative Execution and Memory Disambiguation . . . 3-45
3.6.4 Store Forwarding . . . 3-45
3.6.4.1 Store-to-Load-Forwarding Restriction on Size and Alignment . . . 3-46
3.6.4.2 Store-Forwarding Restriction on Data Availability . . . 3-49
3.6.5 Data Layout Optimizations . . . 3-50
3.6.6 Stack Alignment . . . 3-52
3.6.7 Capacity Limits and Aliasing in Caches . . . 3-53
3.6.8 Mixing Code and Data . . . 3-53
3.6.8.1 Self-Modifying Code (SMC) . . . 3-54
3.6.8.2 Position Independent Code . . . 3-54
3.6.9 Write Combining . . . 3-54
3.6.10 Locality Enhancement . . . 3-55
3.6.11 Non-Temporal Store Bus Traffic . . . 3-56
3.7 PREFETCHING . . . 3-57
3.7.1 Hardware Instruction Fetching and Software Prefetching . . . 3-57
3.7.2 Hardware Prefetching for First-Level Data Cache . . . 3-58
3.7.3 Hardware Prefetching for Second-Level Cache . . . 3-60
3.7.4 Cacheability Instructions . . . 3-60
3.7.5 REP Prefix and Data Movement . . . 3-61
3.7.6 Enhanced REP MOVSB and STOSB Operation . . . 3-63
3.7.6.1 Fast Short REP MOVSB . . . 3-63
3.7.6.2 Memcpy Considerations . . . 3-63
3.7.6.3 Memmove Considerations . . . 3-65
3.7.6.4 Memset Considerations . . . 3-65
3.8 REP STRING OPERATIONS . . . 3-66
3.8.1 Fast Zero Length REP MOVSB . . . 3-66
3.8.2 Fast Short REP STOSB . . . 3-66
3.8.3 Fast Short REP CMPSB and SCASB . . . 3-66
3.9 FLOATING-POINT CONSIDERATIONS . . . 3-66
3.9.1 Guidelines for Optimizing Floating-Point Code . . . 3-67
3.9.2 Floating-Point Modes and Exceptions . . . 3-67

CONTENTS

PAGE

3.9.2.1 Floating-Point Exceptions 3-67
3.9.2.2 Dealing with Floating-Point Exceptions in x87 FPU Code 3-68
3.9.2.3 Floating-Point Exceptions in SSE/SSE2/SSE3 Code 3-68
3.9.3 Floating-Point Modes 3-69
3.9.3.1 Rounding Mode 3-69
3.9.3.2 Precision 3-71
3.9.4 x87 vs. Scalar SIMD Floating-Point Trade-Offs 3-71
3.9.4.1 Scalar Intel® SSE/Intel® SSE2 3-72
3.9.4.2 Transcendental Functions 3-72
3.10 MAXIMIZING PCIE PERFORMANCE 3-72
3.10.1 Optimizing PCIe Performance for Accesses Toward Coherent Memory and MMIO Regions (P2P) 3-73
3.11 SCALABILITY WITH CONTENDED LINE ACCESS IN 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS 3-73
3.11.1 Causes of Performance Bottlenecks 3-73
3.11.2 Performance Bottleneck Detection 3-73
3.11.3 Solutions for Performance Bottlenecks 3-75
3.11.4 Case Study: SysBench/MariaDB 3-76
3.11.5 Scalability With False Sharing 3-77
3.11.5.1 Causes of False Sharing 3-77
3.11.5.2 Detecting False Sharing 3-77
3.11.5.3 Fixing False Sharing and Additional Resources 3-78
3.11.5.4 Case Study: DeathStarBench/hotelReservation 3-79
3.11.6 Instruction Sequence Slowdowns 3-80
3.11.6.1 Causes of Instruction Sequence Slowdowns 3-80
3.11.6.2 Detecting Instruction Sequence Slowdowns 3-81
3.11.6.3 Fixing Instruction Sequence Slowdowns 3-81
3.11.7 Misprediction for Branches >2GB 3-82
3.11.7.1 Causes of Branch Misprediction >2GB 3-82
3.11.7.2 Detecting Branch Mispredictions >2GB 3-82
3.11.7.3 Fixing Branch Mispredictions >2GB 3-83
3.12 OPTIMIZING COMMUNICATION WITH PCI DEVICES ON INTEL® 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS 3-84
3.12.1 Signaling Devices with Direct Move 3-84
3.12.1.1 MOVDIR64B: Additional Considerations 3-85
3.12.1.2 Streaming Data 3-85
3.13 SYNCHRONIZATION 3-85
3.13.1 User-Level Monitor, User-Level MWAIT, and TPAUSE 3-85
3.13.1.1 Checking for User-Level Monitor, MWAIT, and TPAUSE Support 3-86
3.13.1.2 User-Level Monitor, User-Level MWAIT, and TPAUSE Operations 3-86
3.13.1.3 Recommended Usage of Monitor, MWAIT, and TPAUSE Operations 3-86

CHAPTER 4
INTEL ATOM® PROCESSOR ARCHITECTURES

4.1 THE CRESTMONT MICROARCHITECTURE 4-1
4.1.1 Crestmont Microarchitecture Overview 4-2
4.1.2 Predict and Fetch 4-3
4.1.3 Dynamic Load Balancing 4-4
4.1.4 Instruction Decode and the On-Demand Instruction Length Decoder (OD-ILD) 4-4
4.1.5 Allocation and Retirement 4-5
4.1.6 The Out-of-Order and Execution Engines 4-5
4.1.7 Cache and Memory Subsystem 4-6
4.1.8 Crestmont New Instruction Support 4-8
4.1.8.1 AVX-NE-CONVERT Instructions 4-8
4.1.8.2 AVX-IFMA 4-8
4.1.8.3 AVX-VNNI-INT8 Instructions 4-8
4.1.9 Legacy Intel® AVX1/Intel® AVX2 Instruction Support 4-8

CONTENTS

PAGE

4.1.9.1 256-bit Permute Operations 4-8
4.1.9.2 256-bit Broadcast with 128-bit Memory Operand 4-9
4.1.9.3 256-bit Insertion, Up-Conversion Instructions with 128-bit Memory Operand 4-9
4.1.9.4 256-bit Variable Blend Instructions 4-9
4.1.9.5 256-bit Vector TEST Instructions 4-10
4.1.9.6 The GATHER Instruction 4-10
4.1.9.7 Masked Load and Store Instructions 4-10
4.1.9.8 ADX Instructions 4-10
4.2 THE TREMONT MICROARCHITECTURE 4-10
4.2.1 Tremont Microarchitecture Overview 4-12
4.2.2 The Front End 4-12
4.2.3 The Out-of-Order and Execution Engines 4-13
4.2.4 Cache and Memory Subsystem 4-14
4.2.5 New Instructions 4-15
4.2.6 Tremont Microarchitecture Power Management 4-15

CHAPTER 5
CODING FOR SIMD ARCHITECTURES

5.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES 5-1
5.1.1 Checking for MMX Technology Support 5-2
5.1.2 Checking for Intel® Streaming SIMD Extensions (Intel® SSE) Support 5-2
5.1.3 Checking for Intel® Streaming SIMD Extensions 2 (Intel® SSE2) Support 5-2
5.1.4 Checking for Intel® Streaming SIMD Extensions 3 (Intel® SSE3) Support 5-3
5.1.5 Checking for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE) Support 5-3
5.1.6 Checking for Intel® SSE4.1 Support 5-4
5.1.7 Checking for Intel® SSE4.2 Support 5-4
5.1.8 Detection of PCLMULQDQ and AESNI Instructions 5-5
5.1.9 Detection of Intel® AVX Instructions 5-5
5.1.10 Detection of VEX-Encoded AES and VPCLMULQDQ 5-7
5.1.11 Detection of F16C Instructions 5-8
5.1.12 Detection of FMA 5-9
5.1.13 Detection of Intel® AVX2 5-10
5.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING 5-11
5.2.1 Identifying Hot Spots 5-13
5.2.2 Determine If Code Benefits by Conversion to SIMD Execution 5-13
5.3 CODING TECHNIQUES 5-13
5.3.1 Coding Methodologies 5-14
5.3.1.1 Assembly 5-15
5.3.1.2 Intrinsics 5-15
5.3.1.3 Classes 5-16
5.3.1.4 Automatic Vectorization 5-17
5.4 STACK AND DATA ALIGNMENT 5-18
5.4.1 Alignment and Contiguity of Data Access Patterns 5-18
5.4.1.1 Using Padding to Align Data 5-18
5.4.1.2 Using Arrays to Make Data Contiguous 5-18
5.4.2 Stack Alignment for 128-bit SIMD Technologies 5-19
5.4.3 Data Alignment for MMX™ Technology 5-20
5.4.4 Data Alignment for 128-bit data 5-20
5.4.4.1 Compiler-Supported Alignment 5-20
5.5 IMPROVING MEMORY UTILIZATION 5-22
5.5.1 Data Structure Layout 5-22
5.5.2 Strip-Mining 5-25
5.5.3 Loop Blocking 5-26
5.6 INSTRUCTION SELECTION 5-28
5.7 TUNING THE FINAL APPLICATION 5-30

CONTENTS

PAGE

CHAPTER 6
OPTIMIZING FOR SIMD INTEGER APPLICATIONS

6.1 GENERAL RULES ON SIMD INTEGER CODE 6-1
6.2 USING SIMD INTEGER WITH X87 FLOATING-POINT 6-2
6.2.1 Using the EMMS Instruction 6-2
6.2.2 Guidelines for Using EMMS Instruction 6-2
6.3 DATA ALIGNMENT 6-3
6.4 DATA MOVEMENT CODING TECHNIQUES 6-5
6.4.1 Unsigned Unpack 6-5
6.4.2 Signed Unpack 6-6
6.4.3 Interleaved Pack with Saturation 6-6
6.4.4 Interleaved Pack without Saturation 6-8
6.4.5 Non-Interleaved Unpack 6-8
6.4.6 Extract Data Element 6-10
6.4.7 Insert Data Element 6-10
6.4.8 Non-Unit Stride Data Movement 6-12
6.4.9 Move Byte Mask to Integer 6-12
6.4.10 Packed Shuffle Word for 64-bit Registers 6-13
6.4.11 Packed Shuffle Word for 128-bit Registers 6-14
6.4.12 Shuffle Bytes 6-14
6.4.13 Conditional Data Movement 6-14
6.4.14 Unpacking/Interleaving 64-bit Data in 128-bit Registers 6-15
6.4.15 Data Movement 6-15
6.4.16 Conversion Instructions 6-15
6.5 GENERATING CONSTANTS 6-15
6.6 BUILDING BLOCKS 6-16
6.6.1 Absolute Difference of Unsigned Numbers 6-16
6.6.2 Absolute Difference of Signed Numbers 6-17
6.6.3 Absolute Value 6-17
6.6.4 Pixel Format Conversion 6-17
6.6.5 Endian Conversion 6-19
6.6.6 Clipping to an Arbitrary Range [High, Low] 6-20
6.6.6.1 Highly Efficient Clipping 6-21
6.6.6.2 Clipping to an Arbitrary Unsigned Range [High, Low] 6-22
6.6.7 Packed Max/Min of Byte, Word and Dword 6-22
6.6.8 Packed Multiply Integers 6-22
6.6.9 Packed Sum of Absolute Differences 6-23
6.6.10 MPSADBW and PHMINPOSUW 6-23
6.6.11 Packed Average (Byte/Word) 6-23
6.6.12 Complex Multiply by a Constant 6-24
6.6.13 Packed 64-bit Add/Subtract 6-24
6.6.14 128-bit Shifts 6-24
6.6.15 PTEST and Conditional Branch 6-25
6.6.16 Vectorization of Heterogeneous Computations across Loop Iterations 6-25
6.6.17 Vectorization of Control Flows in Nested Loops 6-26
6.7 MEMORY OPTIMIZATIONS 6-29
6.7.1 Partial Memory Accesses 6-29
6.7.2 Increasing Bandwidth of Memory Fills and Video Fills 6-30
6.7.2.1 Increasing Memory Bandwidth Using the MOVDQ Instruction 6-31
6.7.2.2 Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page . . 6-31
6.7.2.3 Increasing UC and WC Store Bandwidth by Using Aligned Stores 6-31
6.7.3 Reverse Memory Copy 6-31
6.8 CONVERTING FROM 64-BIT TO 128-BIT SIMD INTEGERS 6-34
6.8.1 SIMD Optimizations and Microarchitectures 6-35
6.8.1.1 Packed SSE2 Integer versus MMX Instructions 6-35
6.9 TUNING PARTIALLY VECTORIZABLE CODE 6-35
6.10 PARALLEL MODE AES ENCRYPTION AND DECRYPTION 6-38
6.10.1 AES Counter Mode of Operation 6-39

CONTENTS

PAGE

6.10.2 AES Key Expansion Alternative 6-47
6.10.3 Enhancement in Haswell Microarchitecture 6-49
6.10.3.1 Advanced Encryption Standard (AES) and Multi-Buffer Cryptographic Throughput 6-49
6.11 LIGHT-WEIGHT DECOMPRESSION AND DATABASE PROCESSING 6-49
6.11.1 Reduced Dynamic Range Datasets 6-50
6.11.2 Compression and Decompression Using SIMD Instructions 6-50

CHAPTER 7
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

7.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE 7-1
7.2 PLANNING CONSIDERATIONS 7-1
7.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT 7-2
7.4 SCALAR FLOATING-POINT CODE 7-2
7.5 DATA ALIGNMENT 7-2
7.5.1 Data Arrangement 7-2
7.5.1.1 Vertical versus Horizontal Computation 7-3
7.5.1.2 Data Swizzling 7-5
7.5.1.3 Data Deswizzling 7-7
7.5.1.4 Horizontal ADD Using SSE 7-9
7.5.2 Use of CVTTPS2PI/CVTTSS2SI Instructions 7-11
7.5.3 Flush-to-Zero and Denormals-are-Zero Modes 7-11
7.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES 7-11
7.6.1 Dot Product and Horizontal SIMD Instructions 7-11
7.6.2 Vector Normalization 7-13
7.6.3 Using Horizontal SIMD Instruction Sets and Data Layout 7-15
7.6.3.1 SOA and Vector Matrix Multiplication 7-17

CHAPTER 8
INT8 DEEP LEARNING INFERENCE

8.1 INTRODUCING INT8 AS DATA TYPE FOR DEEP LEARNING INFERENCE 8-1
8.2 INTRODUCING INTEL® DL BOOST 8-1
8.2.1 Multiply and Add Unsigned and Signed Bytes (VPDPBUSD Instruction) 8-2
8.2.2 Multiply & Add Signed Word Integers (VPDPWSSD Instruction) 8-4
8.3 GENERAL OPTIMIZATIONS 8-4
8.3.1 Memory Layout 8-4
8.3.2 Quantization 8-5
8.3.2.1 Quantization of Weights 8-5
8.3.2.2 Quantization of Activations 8-5
8.3.2.3 Quantizing Negative Activations 8-6
8.3.3 Multicore Considerations 8-6
8.3.3.1 Large Batch (Throughput Workload) 8-6
8.3.3.2 Small Batch (Throughput at Latency Workload) 8-7
8.3.3.3 NUMA 8-7
8.4 CNNS 8-7
8.4.1 Convolutional Layers 8-7
8.4.1.1 Direct Convolution 8-7
8.4.1.2 Convolutional Layers with Low OFM Count 8-14
8.4.2 Post Convolution 8-16
8.4.2.1 Fused Quantization/Dequantization 8-16
8.4.2.2 ReLu 8-17
8.4.2.3 EltWise 8-17
8.4.2.4 Pooling 8-18
8.4.2.5 Pixel Shuffler 8-20
8.5 LSTM NETWORKS 8-22
8.5.1 Fused LSTM Embedding 8-22
8.5.2 Fused post GEMM 8-22

CONTENTS

PAGE

8.5.3 Dynamic Batch Size 8-25
8.5.4 NMT Example: Beam Search Decoder Get Top K 8-25

CHAPTER 9
OPTIMIZING CACHE USAGE

9.1 GENERAL PREFETCH CODING GUIDELINES 9-1
9.2 PREFETCH AND CACHEABILITY INSTRUCTIONS 9-2
9.3 PREFETCH 9-2
9.3.1 Software Data Prefetch 9-2
9.3.2 Prefetch Instructions 9-3
9.3.3 Prefetch and Load Instructions 9-5
9.4 CACHEABILITY CONTROL 9-5
9.4.1 The Non-temporal Store Instructions 9-5
9.4.1.1 Fencing 9-6
9.4.1.2 Streaming Non-temporal Stores 9-6
9.4.1.3 Memory Type and Non-temporal Stores 9-6
9.4.1.4 Write-Combining 9-6
9.4.2 Streaming Store Usage Models 9-7
9.4.2.1 Coherent Requests 9-7
9.4.2.2 Non-coherent requests 9-7
9.4.3 Streaming Store Instruction Descriptions 9-8
9.4.4 The Streaming Load Instruction 9-8
9.4.5 FENCE Instructions 9-8
9.4.5.1 SFENCE Instruction 9-8
9.4.5.2 LFENCE Instruction 9-9
9.4.5.3 MFENCE Instruction 9-9
9.4.6 CLFLUSH Instruction 9-9
9.4.7 CLFLUSHOPT Instruction 9-10
9.5 MEMORY OPTIMIZATION USING PREFETCH 9-12
9.5.1 Software-Controlled Prefetch 9-12
9.5.2 Hardware Prefetch 9-12
9.5.3 Example of Effective Latency Reduction with Hardware Prefetch 9-13
9.5.4 Example of Latency Hiding with S/W Prefetch Instruction 9-14
9.5.5 Software Prefetching Usage Checklist 9-15
9.5.6 Software Prefetch Scheduling Distance 9-16
9.5.7 Software Prefetch Concatenation 9-16
9.5.8 Minimize Number of Software Prefetches 9-17
9.5.9 Mix Software Prefetch with Computation Instructions 9-19
9.5.10 Software Prefetch and Cache Blocking Techniques 9-20
9.5.11 Hardware Prefetching and Cache Blocking Techniques 9-23
9.5.12 Single-Pass versus Multi-Pass Execution 9-24
9.6 MEMORY OPTIMIZATION USING NON-TEMPORAL STORES 9-25
9.6.1 Non-Temporal Stores and Software Write-Combining 9-25
9.6.2 Cache Management 9-26
9.6.2.1 Video Encoder 9-26
9.6.2.2 Video Decoder 9-26
9.6.2.3 Conclusions from Video Encoder and Decoder Implementation 9-27
9.6.2.4 Optimizing Memory Copy Routines 9-27
9.6.2.5 Using the 8-byte Streaming Stores and Software Prefetch 9-28
9.6.2.6 Using 16-byte Streaming Stores and Hardware Prefetch 9-29
9.6.2.7 Performance Comparisons of Memory Copy Routines 9-30
9.6.3 Deterministic Cache Parameters 9-30
9.6.3.1 Cache Sharing Using Deterministic Cache Parameters 9-32
9.6.3.2 Cache Sharing in Single-Core or Multicore 9-32
9.6.3.3 Determine Prefetch Stride 9-32

CONTENTS

PAGE

CHAPTER 10
SUB-NUMA CLUSTERING

10.1 SUB-NUMA CLUSTERING 10-1
10.2 COMPARISON WITH CLUSTER-ON-DIE 10-2
10.3 SNC USAGE 10-2
10.3.1 How to Check NUMA Configuration 10-2
10.3.2 MPI Optimizations for SNC 10-7
10.3.3 SNC Performance Comparison 10-8

CHAPTER 11
MULTICORE AND INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

11.1 PERFORMANCE AND USAGE MODELS 11-1
11.1.1 Multithreading 11-1
11.1.2 Multitasking Environment 11-3
11.2 PROGRAMMING MODELS AND MULTITHREADING 11-3
11.2.1 Parallel Programming Models 11-4
11.2.1.1 Domain Decomposition 11-4
11.2.2 Functional Decomposition 11-4
11.2.3 Specialized Programming Models 11-4
11.2.3.1 Producer-Consumer Threading Models 11-5
11.2.4 Tools for Creating Multithreaded Applications 11-8
11.2.4.1 Programming with OpenMP Directives 11-8
11.2.4.2 Automatic Parallelization of Code 11-8
11.2.4.3 Supporting Development Tools 11-8
11.3 OPTIMIZATION GUIDELINES 11-8
11.3.1 Key Practices of Thread Synchronization 11-9
11.3.2 Key Practices of System Bus Optimization 11-9
11.3.3 Key Practices of Memory Optimization 11-9
11.3.4 Key Practices of Execution Resource Optimization 11-10
11.3.5 Generality and Performance Impact 11-10
11.4 THREAD SYNCHRONIZATION 11-10
11.4.1 Choice of Synchronization Primitives 11-11
11.4.2 Synchronization for Short Periods 11-12
11.4.3 Optimization with Spin-Locks 11-13
11.4.4 Synchronization for Longer Periods 11-13
11.4.4.1 Avoid Coding Pitfalls in Thread Synchronization 11-14
11.4.5 Prevent Sharing of Modified Data and False-Sharing 11-15
11.4.6 Placement of Shared Synchronization Variable 11-16
11.5 SYSTEM BUS OPTIMIZATION 11-17
11.5.1 Conserve Bus Bandwidth 11-17
11.5.2 Understand the Bus and Cache Interactions 11-18
11.5.3 Avoid Excessive Software Prefetches 11-18
11.5.4 Improve Effective Latency of Cache Misses 11-19
11.5.5 Use Full Write Transactions to Achieve Higher Data Rate 11-19
11.6 MEMORY OPTIMIZATION 11-19
11.6.1 Cache Blocking Technique 11-20
11.6.2 Shared-Memory Optimization 11-20
11.6.2.1 Minimize Sharing of Data between Physical Processors 11-20
11.6.2.2 Batched Producer-Consumer Model 11-20
11.6.3 Eliminate 64-KByte Aliased Data Accesses 11-22
11.7 FRONT END OPTIMIZATION 11-22
11.7.1 Avoid Excessive Loop Unrolling 11-22
11.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES 11-22
11.8.1 Topology Enumeration of Shared Resources 11-24
11.8.2 Non-Uniform Memory Access (NUMA) 11-24
11.9 OPTIMIZATION OF OTHER SHARED RESOURCES 11-26
11.9.1 Expanded Opportunity for Intel® HT Optimization 11-26

CONTENTS

PAGE

CHAPTER 12
INTEL® OPTANE™ DC PERSISTENT MEMORY

12.1 MEMORY MODE AND APP-DIRECT MODE 12-1
12.1.1 Memory Mode 12-1
12.1.2 App Direct Mode 12-1
12.1.3 Selecting a Mode 12-2
12.2 DEVICE CHARACTERISTICS OF INTEL® OPTANE™ DC PERSISTENT MEMORY MODULE 12-4
12.2.1 Intel® Optane™ DC Persistent Memory Module Latency 12-4
12.2.2 Read vs. Write Bandwidth 12-4
12.2.3 Number of Threads for Optimal Bandwidth 12-6
12.3 PLATFORM IMPLICATIONS OF HANDLING A SECOND TYPE OF MEMORY 12-8
12.3.1 Multi-Processor Cache Coherence 12-8
12.3.2 Shared Queues in the Memory Hierarchy 12-9
12.4 IMPLEMENTING PERSISTENCE FOR MEMORY 12-9
12.5 POWER CONSUMPTION 12-10
12.5.1 Read-Write Equivalence 12-11
12.5.2 Spatial and Temporal Locality 12-12

CHAPTER 13
64-BIT MODE CODING GUIDELINES

13.1 INTRODUCTION 13-1
13.2 CODING RULES AFFECTING 64-BIT MODE 13-1
13.2.1 Use Legacy 32-Bit Instructions When Data Size Is 32 Bits 13-1
13.2.2 Use Extra Registers to Reduce Register Pressure 13-1
13.2.3 Effective Use of 64-Bit by 64-Bit Multiplication 13-2
13.2.4 Replace 128-bit Integer Division with 128-bit Multiplication 13-2
13.2.5 Sign Extension to Full 64-Bits 13-5
13.3 ALTERNATE CODING RULES FOR 64-BIT MODE 13-5
13.3.1 Use 64-Bit Registers Instead of Two 32-Bit Registers for 64-Bit Arithmetic Result 13-5
13.3.2 Using Software Prefetch 13-6

CHAPTER 14
INTEL® SSE4.2 AND SIMD PROGRAMMING FOR TEXT-PROCESSING/LEXING/PARSING

14.1 INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS 14-1
14.1.1 CRC32 14-4
14.2 USING INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS 14-5
14.2.1 Unaligned Memory Access and Buffer Size Management 14-6
14.2.2 Unaligned Memory Access and String Library 14-6
14.3 INTEL® SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES 14-6
14.3.1 Null Character Identification (Strlen equivalent) 14-7
14.3.2 White-Space-Like Character Identification 14-11
14.3.3 Substring Searches 14-13
14.3.4 String Token Extraction and Case Handling 14-19
14.3.5 Unicode Processing and PCMPxSTRy 14-22
14.3.6 Replacement String Library Function Using Intel® SSE4.2 14-27
14.4 INTEL® SSE4.2-ENABLED NUMERICAL AND LEXICAL COMPUTATION 14-28
14.5 NUMERICAL DATA CONVERSION TO ASCII FORMAT 14-34
14.5.1 Large Integer Numeric Computation 14-48
14.5.1.1 MULX Instruction and Large Integer Numeric Computation 14-48

CONTENTS

PAGE

CHAPTER 15
OPTIMIZATIONS FOR INTEL® AVX, INTEL® AVX2, & INTEL® FMA

15.1 INTEL® AVX INTRINSICS CODING 15-2
15.1.1 Intel® AVX Assembly Coding 15-4
15.2 NON-DESTRUCTIVE SOURCE (NDS) 15-6
15.3 MIXING AVX CODE WITH SSE CODE 15-7
15.3.1 Mixing Intel® AVX and Intel SSE in Function Calls 15-11
15.4 128-BIT LANE OPERATION AND AVX 15-12
15.4.1 Programming With the Lane Concept 15-12
15.4.2 Strided Load Technique 15-13
15.4.3 The Register Overlap Technique 15-16
15.5 DATA GATHER AND SCATTER 15-18
15.5.1 Data Gather 15-18
15.5.2 Data Scatter 15-20
15.6 DATA ALIGNMENT FOR INTEL® AVX 15-22
15.6.1 Align Data to 32 Bytes 15-22
15.6.2 Consider 16-Byte Memory Access when Memory is Unaligned 15-23
15.6.3 Prefer Aligned Stores Over Aligned Loads 15-25
15.7 L1D CACHE LINE REPLACEMENTS 15-25
15.8 4K ALIASING 15-26
15.9 CONDITIONAL SIMD PACKED LOADS AND STORES 15-26
15.9.1 Conditional Loops 15-27
15.10 MIXING INTEGER AND FLOATING-POINT CODE 15-29
15.11 HANDLING PORT 5 PRESSURE 15-32
15.11.1 Replace Shuffles with Blends 15-32
15.11.2 Design Algorithm with Fewer Shuffles 15-34
15.11.3 Perform Basic Shuffles on Load Ports 15-37
15.12 DIVIDE AND SQUARE ROOT OPERATIONS 15-38
15.12.1 Single-Precision Divide 15-40
15.12.2 Single-Precision Reciprocal Square Root 15-42
15.12.3 Single-Precision Square Root 15-44
15.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE 15-46
15.14 HALF-PRECISION FLOATING-POINT CONVERSIONS 15-48
15.14.1 Packed Single-Precision to Half-Precision Conversion 15-48
15.14.2 Packed Half-Precision to Single-Precision Conversion 15-50
15.14.3 Locality Consideration for using Half-Precision FP to Conserve Bandwidth 15-50
15.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES 15-52
15.15.1 Optimizing Throughput with FMA and Floating-Point Add/MUL 15-52
15.15.2 Optimizing Throughput with Vector Shifts 15-54
15.16 AVX2 OPTIMIZATION GUIDELINES 15-55
15.16.1 Multi-Buffering and Intel® AVX2 15-60
15.16.2 Modular Multiplication and AVX2 15-60
15.16.3 Data Movement Considerations 15-60
15.16.3.1 SIMD Heuristics to implement Memcpy() 15-61
15.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB 15-62
15.16.3.3 Memset() Implementation Considerations 15-62
15.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code 15-63
15.16.3.5 256-bit Fetch versus Two 128-bit Fetches 15-63
15.16.3.6 Mixing MULX and AVX2 Instructions 15-64
15.16.4 Considerations for Gather Instructions 15-70
15.16.4.1 Strided Loads 15-73
15.16.4.2 Adjacent Loads 15-75
15.16.5 Intel® AVX2 Conversion Remedy to MMX Instruction Throughput Limitation 15-76

CONTENTS

PAGE

CHAPTER 16
POWER OPTIMIZATION FOR MOBILE USAGES

16.1 OVERVIEW. 16-1
16.2 MOBILE USAGE SCENARIOS. 16-1
16.2.1 Intelligent Energy Efficient Software 16-2
16.3 ACPI C-STATES. 16-3
16.3.1 Processor-Specific C4 and Deep C4 States 16-4
16.3.2 Processor-Specific Deep C-States and Intel® Turbo Boost Technology 16-5
16.3.3 Processor-Specific Deep C-States for Sandy Bridge Microarchitecture 16-5
16.3.4 Intel® Turbo Boost Technology 2.0 16-6
16.4 GUIDELINES FOR EXTENDING BATTERY LIFE 16-6
16.4.1 Adjust Performance to Meet Quality of Features 16-6
16.4.2 Reducing Amount of Work 16-7
16.4.3 Platform-Level Optimizations. 16-7
16.4.4 Handling Sleep State Transitions 16-8
16.4.5 Using Enhanced Intel SpeedStep® Technology. 16-8
16.4.6 Enabling Intel® Enhanced Deeper Sleep 16-9
16.4.7 Multicore Considerations 16-10
16.4.7.1 Enhanced Intel SpeedStep® Technology 16-10
16.4.7.2 Thread Migration Considerations 16-10
16.4.7.3 Multicore Considerations for C-States 16-11
16.5 TUNING SOFTWARE FOR INTELLIGENT POWER CONSUMPTION 16-12
16.5.1 Reduction of Active Cycles 16-12
16.5.1.1 Multithreading to Reduce Active Cycles. 16-12
16.5.1.2 Vectorization. 16-13
16.5.2 PAUSE and Sleep(0) Loop Optimization 16-14
16.5.3 Spin-Wait Loops. 16-15
16.5.4 Using Event Driven Service Instead of Polling in Code. 16-15
16.5.5 Reducing Interrupt Rate 16-15
16.5.6 Reducing Privileged Time 16-15
16.5.7 Setting Context Awareness in the Code 16-16
16.5.8 Saving Energy by Optimizing for Performance 16-17
16.6 PROCESSOR SPECIFIC POWER MANAGEMENT OPTIMIZATION FOR SYSTEM SOFTWARE 16-17
16.6.1 Power Management Recommendation of Processor-Specific Inactive State Configurations . . 16-17
16.6.1.1 Balancing Power Management and Responsiveness of Inactive To Active State Transitions 16-19

CHAPTER 17
SOFTWARE OPTIMIZATION FOR INTEL® AVX-512 INSTRUCTIONS

17.1 BASIC INTEL® AVX-512 VS. INTEL® AVX2 CODING 17-2
17.1.1 Intrinsic Coding 17-3
17.1.2 Assembly Coding 17-5
17.2 MASKING. 17-7
17.2.1 Masking Example. 17-8
17.2.2 Masking Cost 17-12
17.2.3 Masking vs. Blending. 17-13
17.2.4 Nested Conditions / Mask Aggregation. 17-15
17.2.5 Memory Masking Microarchitecture Improvements. 17-16
17.2.6 Peeling and Remainder Masking 17-16
17.3 FORWARDING AND UNMASKED OPERATIONS 17-19
17.4 FORWARDING AND MEMORY MASKING 17-19
17.5 DATA COMPRESS 17-20
17.5.1 Data Compress Example 17-21
17.6 DATA EXPAND. 17-24
17.6.1 Data Expand Example 17-25
17.7 TERNARY LOGIC 17-27
17.7.1 Ternary Logic Example 1 17-27

CONTENTS

PAGE

17.7.2 Ternary Logic Example 2 17-29
17.8 NEW SHUFFLE INSTRUCTIONS 17-31
17.8.1 Two Source Permute Example 17-32
17.9 BROADCAST 17-34
17.9.1 Embedded Broadcast 17-34
17.9.2 Broadcast Executed on Load Ports 17-34
17.10 EMBEDDED ROUNDING 17-35
17.10.1 Static Rounding Mode 17-36
17.11 SCATTER INSTRUCTION 17-37
17.11.1 Data Scatter Example 17-38
17.12 STATIC ROUNDING MODES, SUPPRESS-ALL-EXCEPTIONS (SAE) 17-41
17.13 QWORD INSTRUCTION SUPPORT 17-41
17.13.1 QUADWORD Support in Arithmetic Instructions 17-41
17.13.2 QUADWORD Support in Convert Instructions 17-45
17.13.3 QUADWORD Support for Convert with Truncation Instructions 17-45
17.14 VECTOR LENGTH ORTHOGONALITY 17-46
17.15 INTEL® AVX-512 INSTRUCTIONS FOR TRANSCENDENTAL SUPPORT 17-46
17.15.1 VRCP14, VRSQRT14 - Software Sequences for 1/x, x/y, sqrt(x) 17-46
17.15.1.1 Application Examples 17-46
17.15.2 VGETMANT VGETEXP - Vector Get Mantissa and Vector Get Exponent 17-47
17.15.2.1 Application Examples 17-47
17.15.3 VRNDSCALE - Vector Round Scale 17-47
17.15.3.1 Application Examples 17-47
17.15.4 VREDUCE - Vector Reduce 17-48
17.15.4.1 Application Examples 17-48
17.15.5 VSCALEF - Vector Scale 17-48
17.15.5.1 Application Examples 17-48
17.15.6 VFPCLASS - Vector Floating Point Class 17-49
17.15.6.1 Application Examples 17-49
17.15.7 VPERM, VPERMI2, VPERMT2 - Small Table Lookup Implementation 17-49
17.15.7.1 Application Examples 17-49
17.16 CONFLICT DETECTION 17-50
17.16.1 Vectorization with Conflict Detection 17-50
17.16.2 Sparse Dot Product with VPCONFLICT 17-54
17.17 INTEL® AVX-512 VECTOR BYTE MANIPULATION INSTRUCTIONS (VBMI) 17-56
17.17.1 Permute Packet Bytes Elements Across Lanes (VPERMB) 17-56
17.17.2 Two-Source Byte Permute Across Lanes (VPERMI2B, VPERMT2B) 17-58
17.17.3 Select Packed Unaligned Bytes from Quadword Sources (VPMULTISHIFTQB) 17-61
17.18 FMA LATENCY 17-64
17.19 MIXING INTEL® AVX OR INTEL® AVX-512 EXTENSIONS WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) CODE 17-65
17.20 MIXING ZMM VECTOR CODE WITH XMM/YMM 17-66
17.21 SERVERS WITH A SINGLE FMA UNIT 17-67
17.22 GATHER/SCATTER TO SHUFFLE (G2S/STS) 17-71
17.22.1 Gather to Shuffle in Strided Loads 17-71
17.22.2 Scatter to Shuffle in Strided Stores 17-72
17.22.3 Gather to Shuffle in Adjacent Loads 17-73
17.23 DATA ALIGNMENT 17-75
17.23.1 Align Data to 64 Bytes 17-75
17.24 DYNAMIC MEMORY ALLOCATION AND MEMORY ALIGNMENT 17-76
17.25 DIVISION AND SQUARE ROOT OPERATIONS 17-77
17.25.1 Divide and Square Root Approximation Methods 17-78
17.25.2 Divide and Square Root Performance 17-78
17.25.3 Approximation Latencies 17-79
17.25.4 Code Snippets 17-82
17.26 CLDEMOTE 17-88
17.26.1 Producer-Consumer Communication in Software 17-88
17.27 TIPS ON COMPILER USAGE 17-89

CONTENTS

PAGE

CHAPTER 18
INTEL® ADVANCED VECTOR EXTENSIONS 512 - FP16 INSTRUCTION SET FOR INTEL® XEON® PROCESSORS

18.1 INTRODUCTION . . . 18-1
18.2 TERM DEFINITIONS . . . 18-1
18.3 OVERVIEW . . . 18-1
18.4 FP16 NUMERIC INSTRUCTIONS . . . 18-2
18.4.1 Data Type Support . . . 18-2
18.4.2 Overview of Intrinsics . . . 18-3
18.4.3 Fundamental Complex-Valued Support . . . 18-4
18.4.4 Using Intel® AVX-512 Bit Masks for Real-Valued Operations . . . 18-5
18.4.5 Using Intel® AVX-512 Bit Masks for Complex-Valued Operations . . . 18-6
18.5 NUMERICS . . . 18-9
18.5.1 Introduction to FP16 Number Format . . . 18-9
18.5.2 Observations on Representing Numbers in FP16 Format . . . 18-10
18.5.3 Numeric Accuracy Guarantees . . . 18-11
18.5.4 Handling Denormal Values . . . 18-12
18.5.5 Embedded Rounding . . . 18-13
18.5.6 Legacy FP16 Data Type Conversion . . . 18-13
18.5.7 FP16 Conversions to and from Other Data Types . . . 18-14
18.5.8 Approximation Instructions and Their Uses . . . 18-14
18.5.8.1 Approximate Reciprocal . . . 18-15
18.5.8.2 Approximate Division . . . 18-15
18.5.8.3 Approximate Reciprocal Square Root . . . 18-16
18.5.9 Approximate Square Root . . . 18-17
18.6 USING EXISTING INTEL® AVX-512 INSTRUCTIONS TO AUGMENT FP16 SUPPORT . . . 18-17
18.6.1 Using Existing Instructions to Extend Intel® AVX-512 FP16 Intrinsics . . . 18-17
18.6.2 Common Convenience Intrinsics . . . 18-18
18.6.3 Using Integer Comparisons for Fast Floating-Point Comparison . . . 18-18
18.7 MATH LIBRARY SUPPORT . . . 18-19

CHAPTER 19
CRYPTOGRAPHY & FINITE FIELD ARITHMETIC ENHANCEMENTS

19.1 VECTOR AES . . . 19-1
19.2 VPCLMULQDQ . . . 19-2
19.3 GALOIS FIELD NEW INSTRUCTIONS . . . 19-2
19.4 INTEGER FUSED MULTIPLY ACCUMULATE OPERATIONS (AVX512_IFMA - VPMADD52) . . . 19-4

CHAPTER 20
INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

20.1 DETECTING INTEL® AMX SUPPORT . . . 20-2
20.2 INTEL® AMX MICROARCHITECTURE OVERVIEW . . . 20-2
20.2.1 Intel® AMX Frequencies . . . 20-2
20.3 INTEL® AMX INSTRUCTIONS THROUGHPUT AND LATENCY . . . 20-3
20.4 DATA STRUCTURE ALIGNMENT . . . 20-3
20.5 GEMMS / CONVOLUTIONS . . . 20-3
20.5.1 Notation . . . 20-3
20.5.2 Tiles in the Intel® AMX Architecture . . . 20-4
20.5.3 B Matrix Layout . . . 20-6
20.5.4 Straightforward GEMM Implementation . . . 20-9
20.5.5 Optimizations . . . 20-10
20.5.5.1 Minimizing Tile Loads . . . 20-10
20.5.5.2 Software Pipelining of Tile Loads and Stores . . . 20-13
20.5.5.3 Optimized GEMM Implementation . . . 20-13
20.5.5.4 Direct Convolution with Intel® AMX . . . 20-15

CONTENTS

PAGE

20.5.5.5 Convolution - Matrix-like Multiplications and Summations Equivalence 20-17
20.5.5.6 Optimized Convolution Implementation 20-19
20.6 CACHE BLOCKING 20-21
20.6.1 Optimized Convolution Implementation with Cache Blocking 20-21
20.7 MINI-BATCHING IN LARGE BATCH INFERENCE 20-24
20.8 NON-TEMPORAL TILE LOADS 20-25
20.8.1 Priority Inversion Scenarios with Temporal Loads 20-26
20.9 USING LARGE TILES IN SMALL CONVOLUTIONS TO MAXIMIZE DATA REUSE 20-28
20.10 HANDLING INCONVENIENTLY-SIZED ACTIVATIONS 20-29
20.11 POST-CONVOLUTION OPTIMIZATIONS 20-29
20.11.1 Post-Convolution Fusion 20-29
20.11.2 Intel® AMX and Intel® AVX-512 Interleaving (SW Pipelining) 20-32
20.11.3 AVOIDING THE H/W OVERHEAD OF FREQUENT OPEN/CLOSE OPERATIONS in Port Five 20-35
20.11.4 Post-Convolution Multiple OFM Accumulation and Efficient Down-Conversion 20-35
20.12 INPUT AND OUTPUT BUFFERS REUSE (DOUBLE BUFFERING) 20-37
20.13 SOFTWARE PREFETCHES 20-39
20.13.1 Software Prefetch for Convolution and GEMM Layers 20-39
20.13.1.1 The Prefetch Strategy 20-39
20.13.1.2 Prefetch Distance 20-39
20.13.1.3 To Prefetch A or Prefetch B? 20-40
20.13.1.4 To Prefetch or Not to Prefetch C? 20-40
20.13.2 Software Prefetch for Embedding Layer 20-40
20.14 HARDWARE PREFETCHER TUNING 20-41
20.15 STORE TO LOAD FORWARDING 20-41
20.16 MATRIX TRANSPOSE 20-42
20.16.1 Flat-to-Flat Transpose of BF16 Data 20-42
20.16.2 VNNI-to-VNNI Transpose 20-46
20.16.3 Flat-to-VNNI Transpose 20-48
20.16.4 Flat-to-VNNI Re-layout 20-51
20.17 MULTI-THREADING CONSIDERATIONS 20-52
20.17.1 Thread Affinity 20-52
20.17.2 Intel® Hyper-Threading Technology (Intel® HT) 20-52
20.17.3 Work Partitioning Between Cores 20-53
20.17.3.1 Partitioning Over M 20-53
20.17.3.2 Partitioning Over N 20-53
20.17.3.3 Partitioning Over K 20-54
20.17.3.4 Memory Bandwidth Implications of Work Partitioning Over Multiple Dimensions 20-55
20.17.4 Recommendation System Example 20-56
20.18 SPARSITY OPTIMIZATIONS FOR INTEL® AMX 20-57
20.19 TILECONFIG/TILERELEASE, CORE C-STATE, AND COMPILER ABI 20-59
20.19.1 ABI 20-59
20.19.2 Intrinsics 20-60
20.19.3 User Interface 20-60
20.19.4 Intel® AMX Intrinsics Example 20-63
20.19.5 Compilation Option 20-65
20.20 INTEL® AMX STATE MANAGEMENT 20-65
20.20.1 Extended Feature Disable (XFD) 20-66
20.20.2 Alternate Signal Handler Stack in Linux Operating System 20-66
20.21 USING INTEL® AMX TO EMULATE HIGHER PRECISION GEMMS 20-66

CHAPTER 21
INTEL® QUICKASSIST TECHNOLOGY (INTEL® QAT)

21.1 SOFTWARE DESIGN GUIDELINES 21-1
21.1.1 Polling vs. Interrupts (If Supported) 21-1
21.1.1.1 Interrupt Mode 21-2
21.1.1.2 Polling Mode 21-2
21.1.1.3 Recommendations 21-3

PAGE

21.1.2 Use of Data Plane (DP) API vs. Traditional API 21-3
21.1.2.1 Batch Submission of Requests Using the Data Plane API 21-3
21.1.3 Synchronous (sync) vs. Asynchronous (async) 21-4
21.1.4 Buffer Lists 21-4
21.1.5 Maximum Number of Concurrent Requests 21-4
21.1.6 Symmetric Crypto Partial Operations 21-5
21.1.7 Reusing Sessions in QAT Environment 21-5
21.1.8 Maximizing QAT Device Utilization 21-5
21.1.9 Best Known Method (BKM) for Avoiding Performance Bottlenecks 21-6
21.1.10 Avoid Data Copies By Using SVM and ATS 21-6
21.1.11 Avoid Page Faults When Using SVM 21-6

CHAPTER 22
USING PERFORMANCE MONITORING EVENTS

22.1 TOP-DOWN ANALYSIS METHOD 22-1
22.1.1 Top-Level 22-2
22.1.2 Frontend Bound 22-4
22.1.3 Backend Bound 22-4
22.1.4 Memory Bound 22-4
22.1.5 Core Bound 22-5
22.1.6 Bad Speculation 22-5
22.1.7 Retiring 22-6
22.1.8 Golden Cove Microarchitecture 22-6
22.1.9 Ice Lake Microarchitecture 22-6
22.1.10 Optane Persistent Memory 22-6
22.1.11 Skylake Microarchitecture 22-6
22.1.11.1 TMA Use Case 1 22-7
22.1.11.2 TMA Use Case 2 22-7
22.2 PERFORMANCE MONITORING AND MICROARCHITECTURE 22-8
22.3 INTEL® XEON® PROCESSOR 5500 SERIES 22-15
22.4 PERFORMANCE ANALYSIS TECHNIQUES FOR INTEL® XEON® PROCESSOR 5500 SERIES 22-16
22.4.1 Cycle Accounting and Uop Flow Analysis 22-17
22.4.1.1 Cycle Drill Down and Branch Mispredictions 22-18
22.4.1.2 Basic Block Drill Down 22-21
22.4.2 Stall Cycle Decomposition and Core Memory Accesses 22-22
22.4.2.1 Measuring Costs of Microarchitectural Conditions 22-22
22.4.3 Core PMU Precise Events 22-23
22.4.3.1 Precise Memory Access Events 22-24
22.4.3.2 Load Latency Event 22-25
22.4.3.3 Precise Execution Events 22-27
22.4.3.4 Last Branch Record (LBR) 22-28
22.4.3.5 Measuring Per-Core Bandwidth 22-33
22.4.3.6 Miscellaneous L1 and L2 Events for Cache Misses 22-34
22.4.3.7 TLB Misses 22-34
22.4.3.8 L1 Data Cache 22-34
22.4.4 Front End Monitoring Events 22-35
22.4.4.1 Branch Mispredictions 22-35
22.4.4.2 Front End Code Generation Metrics 22-35
22.4.5 Uncore Performance Monitoring Events 22-36
22.4.5.1 Global Queue Occupancy 22-36
22.4.5.2 Global Queue Port Events 22-38
22.4.5.3 Global Queue Snoop Events 22-38
22.4.5.4 L3 Events 22-39
22.4.6 Intel QuickPath Interconnect Home Logic (QHL) 22-39
22.4.7 Measuring Bandwidth From the Uncore 22-45
22.5 PERFORMANCE TUNING TECHNIQUES FOR SANDY BRIDGE MICROARCHITECTURE 22-45
22.5.1 Correlating Performance Bottleneck to Source Location 22-45

CONTENTS

PAGE

22.5.2 Hierarchical Top-Down Performance Characterization Methodology and Locating Performance Bottlenecks . . . 22-46
22.5.2.1 Back End Bound Characterization . . . 22-47
22.5.2.2 Core Bound Characterization . . . 22-47
22.5.2.3 Memory Bound Characterization . . . 22-48
22.5.3 Back End Stalls . . . 22-49
22.5.4 Memory Sub-System Stalls . . . 22-50
22.5.4.1 Accounting for Load Latency . . . 22-50
22.5.4.2 Cache-line Replacement Analysis . . . 22-52
22.5.4.3 Lock Contention Analysis . . . 22-52
22.5.4.4 Other Memory Access Issues . . . 22-53
22.5.5 Execution Stalls . . . 22-56
22.5.5.1 Longer Instruction Latencies . . . 22-56
22.5.5.2 Assists . . . 22-57
22.5.6 Bad Speculation . . . 22-57
22.5.6.1 Branch Mispredicts . . . 22-57
22.5.7 Frontend Stalls . . . 22-58
22.5.7.1 Understanding the Micro-op Delivery Rate . . . 22-58
22.5.7.2 Understanding the Sources of the Micro-op Queue . . . 22-59
22.5.7.3 The Decoded ICache . . . 22-61
22.5.7.4 Issues in the Legacy Decode Pipeline . . . 22-62
22.5.7.5 Instruction Cache . . . 22-62
22.6 USING PERFORMANCE EVENTS OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS . . . 22-62
22.6.1 Understanding the Results in a Performance Counter . . . 22-63
22.6.2 Ratio Interpretation . . . 22-63
22.6.3 Notes on Selected Events . . . 22-64
22.7 DRILL-DOWN TECHNIQUES FOR PERFORMANCE ANALYSIS . . . 22-64
22.7.1 Cycle Composition at Issue Port . . . 22-66
22.7.2 Cycle Composition of OOO Execution . . . 22-66
22.7.3 Drill-Down on Performance Stalls . . . 22-67
22.8 EVENT RATIOS FOR INTEL CORE MICROARCHITECTURE . . . 22-68
22.8.1 Clocks Per Instructions Retired Ratio (CPI) . . . 22-68
22.8.1.1 Front End Ratios . . . 22-68
22.8.1.2 Branching and Front End . . . 22-69
22.8.1.3 Stack Pointer Tracker . . . 22-69
22.8.1.4 Macro-fusion . . . 22-69
22.8.1.5 Length Changing Prefix (LCP) Stalls . . . 22-69
22.8.1.6 Self Modifying Code Detection . . . 22-70
22.8.2 Branch Prediction Ratios . . . 22-70
22.8.2.1 Branch Mispredictions . . . 22-70
22.8.2.2 Virtual Tables and Indirect Calls . . . 22-70
22.8.2.3 Mispredicted Returns . . . 22-70
22.8.3 Execution Ratios . . . 22-71
22.8.3.1 Resource Stalls . . . 22-71
22.8.3.2 ROB Read Port Stalls . . . 22-71
22.8.3.3 Partial Register Stalls . . . 22-71
22.8.3.4 Partial Flag Stalls . . . 22-71
22.8.3.5 Bypass Between Execution Domains . . . 22-71
22.8.3.6 Floating-Point Performance Ratios . . . 22-71
22.8.4 Memory Sub-System - Access Conflicts Ratios . . . 22-72
22.8.4.1 Loads Blocked by the L1 Data Cache . . . 22-72
22.8.4.2 4K Aliasing and Store Forwarding Block Detection . . . 22-72
22.8.4.3 Load Block by Preceding Stores . . . 22-72
22.8.4.4 Memory Disambiguation . . . 22-73
22.8.4.5 Load Operation Address Translation . . . 22-73
22.8.5 Memory Sub-System - Cache Misses Ratios . . . 22-73
22.8.5.1 Locating Cache Misses in the Code . . . 22-73
22.8.5.2 L1 Data Cache Misses . . . 22-73

CONTENTS

PAGE

22.8.5.3 L2 Cache Misses 22-73
22.8.6 Memory Sub-system - Prefetching 22-74
22.8.6.1 L1 Data Prefetching 22-74
22.8.6.2 L2 Hardware Prefetching 22-74
22.8.6.3 Software Prefetching 22-74
22.8.7 Memory Sub-system - TLB Miss Ratios 22-74
22.8.8 Memory Sub-system - Core Interaction 22-75
22.8.8.1 Modified Data Sharing 22-75
22.8.8.2 Fast Synchronization Penalty 22-75
22.8.8.3 Simultaneous Extensive Stores and Load Misses 22-75
22.8.9 Memory Sub-system - Bus Characterization 22-75
22.8.9.1 Bus Utilization 22-75
22.8.9.2 Modified Cache Lines Eviction 22-76

APPENDIX A
APPLICATION PERFORMANCE TOOLS

A.1 COMPILERS A-2
A.1.1 Recommended Optimization Settings for Intel® 64 and IA-32 Processors A-2
A.1.2 Vectorization and Loop Optimization A-3
A.1.2.1 Multithreading with OpenMP* A-3
A.1.2.2 Automatic Multithreading A-3
A.1.3 Inline Expansion of Library Functions (/Oi, /Oi-) A-3
A.1.4 Interprocedural and Profile-Guided Optimizations A-3
A.1.4.1 Interprocedural Optimization (IPO) A-4
A.1.4.2 Profile-Guided Optimization (PGO) A-4
A.1.5 Intel® Cilk™ Plus A-4
A.2 PERFORMANCE LIBRARIES A-4
A.2.1 Intel® Integrated Performance Primitives (Intel® IPP) A-5
A.2.2 Intel® Math Kernel Library (Intel® MKL) A-5
A.2.3 Intel® Threading Building Blocks (Intel® TBB) A-5
A.2.4 Benefits Summary A-5
A.3 PERFORMANCE PROFILERS A-6
A.3.1 Intel® VTune™ Amplifier XE A-6
A.3.1.1 Hardware Event-Based Sampling Analysis A-6
A.3.1.2 Algorithm Analysis A-6
A.3.1.3 Platform Analysis A-7
A.4 THREAD AND MEMORY CHECKERS A-7
A.4.1 Intel® Inspector A-7
A.5 VECTORIZATION ASSISTANT A-7
A.5.1 Intel® Advisor A-7
A.6 CLUSTER TOOLS A-7
A.6.1 Intel® Trace Analyzer and Collector A-7
A.6.1.1 MPI Performance Snapshot A-8
A.6.2 Intel® MPI Library A-8
A.6.3 Intel® MPI Benchmarks A-8
A.7 INTEL® COMMUNITIES A-8

APPENDIX B
RUNTIME PERFORMANCE OPTIMIZATION BLUEPRINT: INTEL® ARCHITECTURE OPTIMIZATION WITH LARGE CODE PAGES

B.1 OVERVIEW B-1
B.1.1 TLBs and Stalls B-1
B.1.2 Large Pages B-2
B.2 DIAGNOSING THE PROBLEM B-2
B.2.1 ITLB Misses B-2
B.2.2 Measuring the ITLB Miss Stall B-5

CONTENTS

PAGE

B.2.3 Source of ITLB Misses B-6
B.3 SOLUTION B-7
B.3.1 Linux* and Large Pages B-7
B.3.2 Large Pages for .text B-7
B.3.3 Reference Code B-8
B.3.4 Large Pages for the Heap B-9
B.4 SOLUTION INTEGRATION B-10
B.4.1 V8 Integration with the Reference Implementation B-10
B.4.2 JAVA JVM Integration with the Reference Implementation B-10
B.5 LIMITATIONS B-11
B.6 CASE STUDY B-11
B.6.1 Ghost.js Workload B-12
B.6.2 Web Tooling Workload B-12
B.6.2.1 Node Version B-12
B.6.2.2 Web Tooling B-13
B.6.2.3 Comparing Clear Linux* OS and Ubuntu* B-13
B.6.3 MediaWiki Workload B-13
B.6.4 Visualization of Benefits B-14
B.6.4.1 Precise Events B-14
B.6.4.2 Visualizing Precise ITLB Miss B-14
B.7 SUMMARY B-17
B.8 TEST CONFIGURATION DETAILS B-17
B.9 ADDITIONAL REFERENCES B-19

TABLES

PAGE

Table 1-1. Intel Processors Organized by Microarchitecture 1-2
Table 1-2. Term Definitions 1-6
Table 1-3. Additional References in and Beyond this Document 1-7
Table 2-1. MOV+OP Macro-Fusion Example 2-3
Table 2-2. LD+OP Macro-Fusion Example 2-3
Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture 2-12
Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions 2-13
Table 2-5. Bypass Delay Between Producer and Consumer Micro-Ops 2-14
Table 2-6. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture 2-19
Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions. 2-19
Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops. 2-20
Table 2-9. Cache Parameters of the Ice Lake Client Microarchitecture 2-21
Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture 2-21
Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture . . 2-27
Table 2-12. Maximum Intel® Turbo Boost Technology Core Frequency Levels 2-30
Table 2-13. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture. 2-34
Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions 2-34
Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops. 2-35
Table 2-16. Cache Parameters of the Skylake Client Microarchitecture 2-36
Table 2-17. TLB Parameters of the Skylake Client Microarchitecture 2-37
Table 3-1. Macro-Fusible Instructions in the Sandy Bridge Microarchitecture 3-13
Table 3-2. Macro-Fusible Instructions in the Haswell Microarchitecture 3-13
Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction. 3-27
Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture 3-48
Table 3-5. Relative Performance of Memcpy() Using Enhanced REP MOVSB and STOSB Vs. 128-bit AVX3-65
Table 3-6. Effect of Address Misalignment on Memcpy() Performance 3-65
Table 3-7. Intel Processor CPU RP Device IDs for Processors Optimizing PCIe Performance 3-73
Table 3-8. Samples: 365K of Events 'anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}', Event Count (a--r0x): 679008523-74
Table 3-9. Shared Data Cache Line Table 3-79
Table 3-10. Shared Cache Line Distribution Pareto 3-79
Table 3-11. False Sharing Improvements 3-80
Table 3-12. Instruction Sequence Mixing VEX on the Sapphire Rapids and Ice Lake Server Microarchitectures3-81
Table 3-13. Fixed Instruction Sequence with Improved Performance on Sapphire Rapids Microarchitecture3-82
Table 3-14. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction 3-83
Table 4-1. Paging Cache Parameters of the Crestmont Microarchitecture 4-7
Table 4-2. Dispatch Port and Execution Stacks of the Tremont Microarchitecture 4-14
Table 4-3. Cache Parameters of the Tremont Microarchitecture 4-15
Table 6-1. PSHUF Instruction Byte Encoding of Individual Words 6-13
Table 7-1. SoA Form of Representing Vertices Data 7-4
Table 9-1. Implementation Details of Prefetch Hint Instructions for Intel® Core™ Duo, Intel® Core™ 2, and Intel® Atom® Processors9-4
Table 9-2. Details of Prefetch Hint Instructions for Processors Based on Skylake Microarchitectures9-4
Table 9-3. Implementation Details of Prefetch Hint Instructions for Intel® Xeon® Scalable Family of Processors9-4
Table 9-4. Software Prefetching Considerations into Strip-mining Code 9-23
Table 9-5. Deterministic Cache Parameters Leaf 9-31
Table 11-1. Properties of Synchronization Objects 11-11
Table 11-2. Design-Time Resource Management Choices 11-23
Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations 11-26

TABLES

PAGE

Table 12-1. Latencies for Accessing Intel® Optane™ DC Persistent Memory Modules 12-4
Table 12-2. Bandwidths per DIMM for Intel® Optane™ DC Persistent Memory Modules and DRAM 12-5
Table 14-1. Intel® SSE4.2 String/Text Instructions Compare Operation on N-Elements 14-3
Table 14-2. Intel® SSE4.2 String/Text Instructions Unary Transformation on IntRes1. 14-3
Table 14-3. Intel® SSE4.2 String/Text Instructions Output Selection Imm[6] 14-3
Table 14-4. SSE4.2 String/Text Instructions Element-Pair Comparison Definition 14-4
Table 14-5. SSE4.2 String/Text Instructions Eflags Behavior 14-4
Table 15-1. Features Between 256-bit Intel® AVX, 128-bit Intel® AVX, and Legacy Intel® SSE Extensions 15-2
Table 15-2. State Transitions of Mixing AVX and SSE Code 15-10
Table 15-3. Approximate Magnitude of Intel® AVX—Intel® SSE Transition Penalties in Different Microarchitectures. 15-10
Table 15-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code. 15-12
Table 15-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture 15-39
Table 15-6. Single-Precision Divide and Square Root Alternatives 15-40
Table 15-7. Comparison of AOS to SOA with Strided Access Pattern 15-74
Table 15-8. Comparison of Indexed AOS to SOA Transformation 15-76
Table 16-1. ACPI C-State-Type Mappings to Processor Specific C-State of the Sandy Bridge Microarchitecture 16-5
Table 16-2. C-State Total Processor Exit Latency for Client Systems with Slow VR 16-18
Table 16-3. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Fast VR 16-18
Table 16-4. C-State Core-Only Exit Latency for Client Systems with Slow VR 16-19
Table 16-5. POWER_CTL MSR in Processors Based on Sandy Bridge Microarchitecture. 16-19
Table 17-1. Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations 17-1
Table 17-2. Cache Comparison Between Skylake Server Microarchitecture and Broadwell Microarchitecture 17-16
Table 17-3. Static Rounding Mode Functions 17-36
Table 17-4. Software Sequence and Hardware Scatter 17-39
Table 17-5. Vector Quadword Extensions 17-45
Table 17-6. Scalar Quadword Extensions. 17-45
Table 17-7. Vector Quadword Extensions 17-45
Table 17-8. Scalar Quadword Extensions. 17-45
Table 17-9. FMA Unit Latency. 17-64
Table 17-10. Data Alignment Effects on SAXPY Performance vs. Speedup Value 17-76
Table 17-11. Skylake Microarchitecture Recommendations for DIV/SQRT-Based Operations (Single-Precision) 17-78
Table 17-12. Skylake Microarchitecture Recommendations for DIV/SQRT-Based Operations (Double-Precision) 17-78
Table 17-13. 256-bit Intel AVX2 Divide and Square Root Instruction Performance. 17-79
Table 17-14. 512-bit Intel AVX-512 Divide and Square Root Instruction Performance 17-79
Table 17-15. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Single Precision 17-80
Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision 17-80
Table 18-1. Supported FP16 Data Types 18-2
Table 18-2. Example Intrinsic Names 18-3
Table 18-3. Conjugation Instructions 18-5
Table 18-4. Useful or Interesting FP16 Numbers. 18-11
Table 18-5. Conjugation Instructions 18-13
Table 18-6. Conjugation Instructions 18-18
Table 20-1. Intel® AMX-Related Links. 20-1
Table 20-2. Intel® AMX Instruction Throughput and Latency 20-3
Table 20-3. Five Loops in Example 20-20 20-25

TABLES

PAGE

Table 20-4. Accessed Data Sizes: Scenario One 20-26
Table 20-5. Accessed Data Sizes: Scenario Two 20-26
Table 20-6. Accessed Data Sizes Extended to Blocked Code 20-27
Table 20-7. Loading 32 Quarter-Cache Lines into 8 ZMM Registers 20-45
Table 20-8. Loading Eight Quarter-Cache Lines into Two ZMM Registers 20-48
Table 20-9. BF16 Flat-to-VNNI Transpose 20-50
Table 20-10. A Simple Partition of Work Between Three Threads 20-55
Table 20-11. An Optimized Partition of Work Between Three Threads 20-56
Table 22-1. Performance Monitoring Taxonomy 22-9
Table 22-2. Cycle Accounting and Micro-ops Flow Recipe 22-17
Table 22-3. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow 22-18
Table 22-4. Cycle Accounting of Wasted Work Due to Misprediction 22-19
Table 22-5. Cycle Accounting of Instruction Starvation 22-20
Table 22-6. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow 22-21
Table 22-7. Approximate Latency of L2 Misses of Intel Xeon Processor 5500 22-23
Table 22-8. Load Latency Event Programming 22-25
Table 22-9. Data Source Encoding for Load Latency PEBS Record 22-26
Table 22-10. Core PMU Events to Drill Down L2 Misses 22-31
Table 22-11. Core PMU Events for Super Queue Operation 22-31
Table 22-12. Core PMU Event to Drill Down OFFCore Responses 22-31
Table 22-13. OFFCORE_RSP_0 MSR Programming 22-32
Table 22-14. Common Request and Response Types for OFFCORE_RSP_0 MSR 22-32
Table 22-15. Uncore PMU Events for Occupancy Cycles 22-38
Table 22-16. Common QHL Opcode Matching Facility Programming 22-40
Table A-1. Recommended Processor Optimization Options A-2
Table B-1. Core TLB Structure Size and Organization Across Multiple Intel Product Generations B-2
Table B-2. Calculating ITLB Miss Stall for Ghost.js B-3
Table B-3. ITLB MPKI and Executable Sizes Across Various Workloads B-5
Table B-4. Key Metrics for Ghost.js With and Without Large Pages B-12
Table B-5. Key Metrics for Web Tooling across Clear Linux and Ubuntu 18.04 . . . . . B-13
Table B-6. Key Metrics for MediaWiki Workload on HHVM B-14
Table B-7. Precise Front-end Events for ITLB Misses B-14
Table B-8. System Details B-17
Table B-9. Processor Information B-18
Table B-10. Kernel Vulnerability Status B-18

FIGURES

PAGE

Figure 2-1. Layout of Array of Pointers Prefetchers 2-4
Figure 2-2. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture 2-11
Figure 2-3. Processor Front End of the Golden Cove Microarchitecture 2-12
Figure 2-4. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture 2-17
Figure 2-5. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture 2-26
Figure 2-6. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures 2-28
Figure 2-7. Mixed Workloads 2-30
Figure 2-8. LINPACK Performance 2-31
Figure 2-9. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture 2-33
Figure 2-10. Intel® Hyper-Threading Technology on an SMP System 2-41
Figure 2-11. Typical SIMD Operations 2-43
Figure 2-12. SIMD Instruction Register Usage 2-44
Figure 3-1. INT Execution Ports Within the Processor Core Pipeline 3-32
Figure 3-2. Generic Program Flow of Partially Vectorized Code 3-36
Figure 3-3. Memcpy Performance Comparison for Lengths up to 2KB 3-64
Figure 3-4. MariaDB - CHA % Cycles Fast Asserted 3-77
Figure 4-1. Processor Core Pipeline Functionality of the Crestmont Microarchitecture 4-2
Figure 4-2. Front-End Pipeline Functionality of the Crestmont Microarchitecture 4-3
Figure 4-3. Execution Pipeline Functionality of the Crestmont Microarchitecture 4-5
Figure 4-4. Processor Core Pipeline Functionality of the Tremont Microarchitecture 4-12
Figure 5-1. General Procedural Flow of Application Detection of Intel® AVX 5-6
Figure 5-2. General Procedural Flow of Application Detection of Float-16 5-8
Figure 5-3. Converting to Intel® Streaming SIMD Extensions Chart 5-12
Figure 5-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-Offs 5-14
Figure 5-5. Loop Blocking Access Pattern 5-28
Figure 6-1. PACKSSDW mm, mm/mm64 Instruction 6-7
Figure 6-2. Interleaved Pack with Saturation 6-7
Figure 6-3. Result of Non-Interleaved Unpack Low in MM0 6-9
Figure 6-4. Result of Non-Interleaved Unpack High in MM1 6-9
Figure 6-5. PEXTRW Instruction 6-10
Figure 6-6. PINSRW Instruction 6-11
Figure 6-7. PMOVSMKB Instruction 6-13
Figure 6-8. Data Alignment of Loads and Stores in Reverse Memory Copy 6-32
Figure 6-9. A Technique to Avoid Cacheline Split Loads in Reverse Memory Copy Using Two Aligned Loads 6-34
Figure 7-1. Homogeneous Operation on Parallel Data Elements 7-3
Figure 7-2. Horizontal Computation Model 7-3
Figure 7-3. Dot Product Operation 7-4
Figure 7-4. Horizontal Add Using MOVHLPS/MOVLHPS 7-9
Figure 8-1. VPMADDUBSW + VPMADDWD + VPADDD Fused into VPDPBUSD (3x Peak Ops on Server Architectures, 2x Peak Ops on Client Architectures) 8-2
Figure 8-2. Matrix Layout, Inputs and Outputs 8-8
Figure 8-3. Transformed Weights 8-8
Figure 8-4. Convolution Operation 8-9
Figure 8-5. Matrix Multiplications and Summations 8-9
Figure 8-6. 3-Tier Flexible 2D Blocking 8-10
Figure 8-7. 3-Tier Flexible 2D Blocking Loops 8-11
Figure 8-8. Standard vs Optimized vs. Low OFM Optimized Data Layouts 8-14
Figure 8-9. Dynamic Batch Size 8-25
Figure 8-10. Find Top 16 Values in Some Input 8-25
Figure 9-1. CLFLUSHOPT versus CLFLUSH In Sky Lake Microarchitecture 9-11
Figure 9-2. Effective Latency Reduction as a Function of Access Stride 9-14
Figure 9-3. Memory Access Latency and Execution Without Prefetch 9-14
Figure 9-4. Memory Access Latency and Execution With Prefetch 9-15

FIGURES

PAGE

Figure 9-5. Prefetch and Loop Unrolling 9-18
Figure 9-6. Memory Access Latency and Execution With Prefetch. 9-18
Figure 9-7. Spread Prefetch Instructions. 9-19
Figure 9-8. Cache Blocking – Temporally Adjacent and Non-adjacent Passes 9-20
Figure 9-9. Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops 9-21
Figure 9-10. Single-Pass vs. Multi-Pass 3D Geometry Engines 9-25
Figure 10-1. Example of SNC Configuration 10-1
Figure 10-2. NUMA Disabled 10-5
Figure 10-3. SNC Off 10-6
Figure 10-4. SNC On 10-7
Figure 10-5. Domain Example with One MPI Process Per Domain 10-8
Figure 11-1. Amdahl's Law and MP Speed-up. 11-2
Figure 11-2. Single-threaded Execution of Producer-consumer Threading Model 11-5
Figure 11-3. Execution of Producer-consumer Threading Model on a Multicore Processor. 11-5
Figure 11-4. Interlaced Variation of the Producer Consumer Model 11-7
Figure 11-5. Batched Approach of Producer Consumer Model. 11-21
Figure 12-1. In App Direct Mode, Data on the Intel® Optane™ DC Persistent Memory Module is Accessed Directly with Loads and Stores 12-2
Figure 12-2. Decision Flow for Determining When to Use Intel® Optane™ DC Persistent Memory Module vs. DRAM 12-3
Figure 12-3. Loaded Latency Curves for One Intel® Optane™ DC Persistent Memory Module DIMM: Sequential Traffic (Left) and Random Traffic (Right). 12-5
Figure 12-4. Number of Threads vs. Bandwidth 12-6
Figure 12-5. Combining with Two Cores 12-7
Figure 12-6. Combining with Four Cores 12-7
Figure 12-7. Combining with Eight Cores 12-8
Figure 12-8. PMDK vs. MSYNC Flushing Times 12-10
Figure 12-9. Bandwidth vs. Power Consumption 12-10
Figure 12-10. Read-Write Equivalence for Intel® Optane™ DC Persistent Memory Module DIMMs within Different Power Budgets 12-11
Figure 12-11. Bandwidth Available to Software when There is No Locality at 256B Granularity. 12-12
Figure 14-1. Intel® SSE4.2 String/Text Instruction Immediate Operand Control 14-2
Figure 14-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search 14-13
Figure 14-3. Intel® SSE4.2 Speedup of SubString Searches 14-19
Figure 14-4. Compute Four Remainders of Unsigned Short Integer in Parallel 14-37
Figure 15-1. Intel® AVX—Intel® SSE Transitions in Earlier Microarchitectures 15-9
Figure 15-2. Intel® AVX- Intel® SSE Transitions in the Skylake Microarchitecture. 15-9
Figure 15-3. Source Location of Each Destination Element in a 128-bit Lane 15-12
Figure 15-4. Gather Operation 15-18
Figure 15-5. Using VPERM2F128 to Swap Elements Between Lanes 15-35
Figure 15-6. Step One of 8 x 8 Matrix Transpose Using VINSERTF128 15-35
Figure 15-7. Intel AVX Implementation of the Complex Multiply Algorithm 15-37
Figure 15-8. SSE Implementation Description. 15-46
Figure 15-9. Intel® AVX Implementation of the Array Sub Sums Algorithm. 15-47
Figure 15-10. 4x4 Image Block Transformation 15-56
Figure 15-11. Throughput Comparison of Gather Instructions 15-72
Figure 15-12. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture. 15-73
Figure 16-1. Performance History and State Transitions 16-2
Figure 16-2. Active Time Versus Halted Time of a Processor 16-3
Figure 16-3. Application of C-States to Idle Time 16-4
Figure 16-4. Profiles of Coarse Task Scheduling and Power Consumption. 16-9
Figure 16-5. Thread Migration in a Multicore Processor 16-11

FIGURES

PAGE

Figure 16-6. Progression to Deeper Sleep 16-11
Figure 16-7. Energy Savings Due to Performance Optimization 16-13
Figure 16-8. Energy Savings Due to Vectorization 16-13
Figure 16-9. Energy-Saving Comparison of Synchronization Primitives 16-16
Figure 16-10. Power-Saving Comparison of Power-Source-Aware Frame Rate Configurations 16-17
Figure 17-1. Cartesian Rotation 17-3
Figure 17-2. Mask Move When Using Merging Masking 17-9
Figure 17-3. Mask Move Operation When Using Merging Masking 17-9
Figure 17-4. Result of Execution with Zeroing Masking 17-9
Figure 17-5. Data Forwarding Cases 17-19
Figure 17-6. Data Compress Operation 17-20
Figure 17-7. Data Expand Operation 17-25
Figure 17-8. Ternary Logic Example 1 Truth Table 17-27
Figure 17-9. Ternary Logic Example 2 Truth Table 17-30
Figure 17-10. VPERMI2PS Instruction Operation 17-31
Figure 17-11. Two-Source Permute Instructions in a Matrix Transpose Operation 17-32
Figure 17-12. VSCATTERDPD Instruction Operation 17-38
Figure 17-13. VPCONFLICTD Instruction Execution 17-51
Figure 17-14. VPCONFLICTD Merging Process 17-51
Figure 17-15. VPCONFLICTD Permute Control 17-52
Figure 17-16. VPCONFLICTD ZMM2 Result 17-54
Figure 17-17. Sparse Vector Example 17-54
Figure 17-18. VPERMB Instruction Operation 17-56
Figure 17-19. VPERMI2B Instruction Operation 17-58
Figure 17-20. VPERMT2B Instruction Operation 17-59
Figure 17-21. VPMULTISHIFTQB Instruction Operation 17-61
Figure 17-22. Fast Bypass When All Sources Come from FMA Unit 17-64
Figure 17-23. Mixing Intel AVX Instructions or Intel AVX-512 Instructions with Intel SSE Instructions 17-65
Figure 18-1. Layout of a 128-Bit Register Representing Four Complex FP16 (CFP16) Values 18-3
Figure 18-2. A Zero-Masked FP16 Add On Two 128-Bit Vectors 18-5
Figure 18-3. A Masked Complex Multiplication 18-6
Figure 18-4. Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition . . 18-7
Figure 18-5. Comparison Operation Between Two Complex-Valued Vectors 18-8
Figure 18-6. Bit Layout of Three Types of Floating-Point Formats 18-10
Figure 18-7. Landmark Numbers on the Real-Valued FP16 Axis 18-10
Figure 18-8. Heat-map Showing Relative ULP Error for Different Combinations of Divisor and Dividend Value Ranges 18-16
Figure 20-1. Matrix Notation 20-4
Figure 20-2. Intel® AMX Multiplication with Max-sized int8 Tiles 20-5
Figure 20-3. Activations layout 20-16
Figure 20-4. Weights Re-Layout 20-17
Figure 20-5. Convolution-Matrix Multiplication and Summation Equivalence 20-17
Figure 20-6. Matrix-Like Multiplications Part of a Convolution 20-18
Figure 20-7. Batching Execution Using Six Layers with Four Instances Per Thread 20-25
Figure 20-8. A Convolution Example 20-28
Figure 20-9. A Convolution Example with Large Tiles 20-28
Figure 20-10. Using TILEZERO to Solve Performance Degradation 20-35
Figure 20-11. A Conversion Flow of 32-bit Integers to 8-bit Integers 20-36
Figure 20-12. Trivial Deep Learning Topology with Naive Buffer Allocation 20-38
Figure 20-13. Minimal Memory Footprint Buffer Allocation Scheme for Trivial Deep Learning Topology . . . 20-38
Figure 20-14. Flat-to-VNNI Transpose of WORDs Equivalence to Flat-to-Flat Transpose of DWORDs 20-48
Figure 20-15. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the M-Dimension . . 20-53
Figure 20-16. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the N-Dimension . . 20-54
Figure 20-17. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the K-Dimension . . . 20-54

TABLES

PAGE

Figure 20-18. A Recommendation System Multi-Threading Model 20-57
Figure 20-19. Data Expand Operation 20-58
Figure 22-1. General TMA Hierarchy for Out-of-Order Microarchitectures 22-2
Figure 22-2. TMA's Top Level Drill Down Flowchart 22-3
Figure 22-3. TMA Hierarchy and Precise Events in the Skylake Microarchitecture 22-8
Figure 22-4. System Topology Supported by Intel® Xeon® Processor 5500 Series 22-16
Figure 22-5. PMU Specific Event Logic Within the Pipeline 22-18
Figure 22-6. LBR Records and Basic Blocks 22-29
Figure 22-7. Using LBR Records to Rectify Skewed Sample Distribution 22-30
Figure 22-8. RdData Request after LLC Miss to Local Home (Clean Rsp) 22-41
Figure 22-9. RdData Request after LLC Miss to Remote Home (Clean Rsp) 22-41
Figure 22-11. RdData Request after LLC Miss to Local Home (Hitm Response) 22-42
Figure 22-10. RdData Request after LLC Miss to Remote Home (Hitm Response) 22-42
Figure 22-12. RdData Request after LLC Miss to Local Home (Hit Response) 22-43
Figure 22-13. RdInvOwn Request after LLC Miss to Remote Home (Clean Res) 22-43
Figure 22-15. RdInvOwn Request after LLC Miss to Local Home (Hit Res) 22-44
Figure 22-14. RdInvOwn Request after LLC Miss to Remote Home (Hitm Res) 22-44
Figure 22-16. Performance Events Drill-Down and Software Tuning Feedback Loop 22-65
Figure B-1. ITLB and ITLB 4K MPKI Across Runtime Workloads B-4
Figure B-2. measure-perf-metric.sh Tool Usage for Process ID 69772 for 30 Seconds B-6
Figure B-3. Using measure-perf-metric.sh with -r to Determine Where TLB Misses are Coming From B-6
Figure B-4. Commands for Checking Linux* Distribution for THP B-7
Figure B-5. API Calls Provided by the Intel Reference Implementation B-9
Figure B-6. perf Output Will Not Have the Proper Symbols After Large Page Mapping B-11
Figure B-7. Using Perf Record with -e frontend_retired.itlb_miss to Determine ITLB Misses B-15
Figure B-8. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload B-15
Figure B-9. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload B-16
Figure B-10. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload B-16
Figure B-11. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload When Run With Large Pages B-17

EXAMPLES

PAGE

Example 2-1. Class 0 Pseudo-code Snippet 2-7
Example 2-2. Class 1 Pseudo-code Snippet 2-8
Example 2-3. Class 2 Pseudo-code Snippet 2-9
Example 2-4. Class 3 Pseudo-code Snippet 2-9
Example 2-5. Breaking False Dependency through Zero Idiom 2-16
Example 2-6. Considering Stores 2-22
Example 2-7. Rearranging Code to Achieve Store Pairing 2-23
Example 2-8. FSFP Optimization 2-24
Example 2-9. Dynamic Pause Loop Example 2-38
Example 2-10. Contended Locks with Increasing Back-off Example 2-39
Example 3-1. Assembly Code With an Unpredictable Branch 3-5
Example 3-2. Code Optimization to Eliminate Branches 3-5
Example 3-3. Eliminating Branch with CMOV Instruction 3-6
Example 3-4. Static Branch Prediction Algorithm 3-6
Example 3-5. Static Taken Prediction 3-7
Example 3-6. Static Not-Taken Prediction 3-7
Example 3-7. Indirect Branch with Two Favored Targets 3-9
Example 3-8. A Peeling Technique to Reduce Indirect Branch Misprediction 3-10
Example 3-9. Loop Unrolling 3-11
Example 3-10. Macrofusion, Unsigned Iteration Count 3-14
Example 3-12. Macrofusion, Signed Variable 3-15
Example 3-11. Macrofusion, If Statement 3-15
Example 3-13. Macrofusion, Signed Comparison 3-16
Example 3-14. Additional Macrofusion Benefit in the Sandy Bridge Microarchitecture 3-16
Example 3-15. Avoiding False LCP Delays with 0xF7 Group Instructions 3-18
Example 3-16. Independent Two-Operand LEA Example 3-22
Example 3-17. Alternative to Three-Operand LEA 3-22
Example 3-18. Examples of 512-bit Additions 3-23
Example 3-19. Clearing Register to Break Dependency While Negating Array Elements 3-26
Example 3-20. Spill Scheduling Code 3-28
Example 3-21. Zero-Latency MOV Instructions 3-29
Example 3-22. Byte-Granular Data Computation Technique 3-30
Example 3-23. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions 3-30
Example 3-24. Avoiding Partial Register Stalls in SIMD Code 3-33
Example 3-25. Avoiding Partial Flag Register Stalls 3-34
Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture 3-34
Example 3-27. Reference Code Template for Partially Vectorizable Program 3-36
Example 3-28. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty 3-37
Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing 3-38
Example 3-30. Stack Optimization Technique to Simplify Parameter Passing 3-38
Example 3-31. Base Line Code Sequence to Estimate Loop Overhead 3-39
Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture 3-41
Example 3-33. Index versus Pointers in Pointer-Chasing Code 3-42
Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy 3-43
Example 3-35. Using XMM Register in Lieu of Memory for Register Spills 3-44
Example 3-36. Loads Blocked by Stores of Unknown Address 3-45
Example 3-37. Situations Showing Small Loads After Large Store 3-47
Example 3-38. Non-forwarding Example of Large Load After Small Store 3-47
Example 3-39. A Non-forwarding Situation in Compiler Generated Code 3-47
Example 3-40. Two Ways to Avoid Non-forwarding Situation in Example 3-39 3-48
Example 3-41. Large and Small Load Stalls 3-48
Example 3-42. Loop-Carried Dependence Chain 3-50
Example 3-43. Rearranging a Data Structure 3-51
Example 3-44. Decomposing an Array 3-51

EXAMPLES

PAGE

Example 3-45. Examples of Dynamical Stack Alignment 3-53
Example 3-46. Instruction Pointer Query Techniques 3-54
Example 3-47. Using Non-Temporal Stores and 64-byte Bus Write Transactions 3-56
Example 3-48. On-temporal Stores and Partial Bus Write Transactions 3-57
Example 3-49. Using DCU Hardware Prefetch 3-58
Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines 3-59
Example 3-51. Technique for Using L1 Hardware Prefetch 3-60
Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination 3-62
Example 3-53. Algorithm to Avoid Changing Rounding Mode 3-70
Example 3-54. Locking Algorithm for the Sapphire Rapids Microarchitecture 3-76
Example 3-55. Perf Annotation for runtime.getempty 3-78
Example 3-56. Padding Insertion in Go Runtime 3-79
Example 3-57. Identifying >2GB Branches 3-83
Example 3-58. Identification of WAITPKG with CPUID 3-86
Example 3-59. Code Snippet in an Asynchronous Example 3-87
Example 5-1. Identification of MMX Technology with CPUID 5-2
Example 5-2. Identification of Intel® SSE with CPUID 5-2
Example 5-3. Identification of Intel® SSE2 with cpuid 5-3
Example 5-4. Identification of Intel® SSE3 with CPUID 5-3
Example 5-5. Identification of SSSE3 with cpuid 5-4
Example 5-6. Identification of Intel® SSE4.1 with CPUID 5-4
Example 5-7. Identification of SSE4.2 with cpuid 5-4
Example 5-8. Detection of AESNI Instructions 5-5
Example 5-9. Detection of PCLMULQDQ Instruction 5-5
Example 5-10. Detection of Intel® AVX Instruction 5-6
Example 5-11. Detection of VEX-Encoded AESNI Instructions 5-7
Example 5-12. Detection of VEX-Encoded AESNI Instructions 5-7
Example 5-13. Simple Four-Iteration Loop 5-15
Example 5-14. Intel® Streaming SIMD Extensions (Intel® SSE) Using Inlined Assembly Encoding 5-15
Example 5-15. Simple Four-Iteration Loop Coded with Intrinsics 5-16
Example 5-16. C++ Code Using the Vector Classes 5-17
Example 5-17. Automatic Vectorization for a Simple Loop 5-17
Example 5-18. C Algorithm for 64-bit Data Alignment 5-20
Example 5-19. AoS Data Structure 5-22
Example 5-20. SoA Data Structure 5-22
Example 5-21. AoS and SoA Code Samples 5-23
Example 5-22. Hybrid SoA Data Structure 5-24
Example 5-23. Pseudo-Code Before Strip Mining 5-25
Example 5-24. Strip Mined Code 5-26
Example 5-25. Loop Blocking 5-26
Example 5-26. Emulation of Conditional Moves 5-29
Example 6-1. Resetting Register Between __m64 and FP Data Types Code 6-3
Example 6-2. FIR Processing Example in C language Code 6-4
Example 6-3. SSE2 and SSSE3 Implementation of FIR Processing Code 6-4
Example 6-4. Zero Extend 16-bit Values into 32 Bits Using Unsigned Unpack Instructions Code 6-5
Example 6-5. Signed Unpack Code 6-6
Example 6-6. Interleaved Pack with Saturation Code 6-7
Example 6-7. Interleaved Pack without Saturation Code 6-8
Example 6-8. Unpacking Two Packed-word Sources in Non-Interleaved Way Code 6-9
Example 6-9. PEXTRW Instruction Code 6-10
Example 6-10. PINSRW Instruction Code 6-11
Example 6-11. Repeated PINSRW Instruction Code 6-11
Example 6-12. Non-Unit Stride Load/Store Using SSE4.1 Instructions 6-12
Example 6-13. Scatter and Gather Operations Using SSE4.1 Instructions 6-12

EXAMPLES

PAGE

Example 6-14. PMOVMSKB Instruction Code 6-13
Example 6-15. Broadcast a Word Across XMM, Using 2 SSE2 Instructions 6-14
Example 6-16. Swap/Reverse Words in an XMM, Using 3 SSE2 Instructions 6-14
Example 6-17. Generating Constants 6-15
Example 6-18. Absolute Difference of Two Unsigned Numbers 6-16
Example 6-19. Absolute Difference of Signed Numbers 6-17
Example 6-20. Computing Absolute Value 6-17
Example 6-21. Basic C Implementation of RGBA to BGRA Conversion 6-18
Example 6-22. Color Pixel Format Conversion Using SSE2 6-18
Example 6-23. Color Pixel Format Conversion Using SSSE3 6-19
Example 6-24. Big-Endian to Little-Endian Conversion 6-20
Example 6-25. Clipping to a Signed Range of Words [High, Low] 6-21
Example 6-26. Clipping to an Arbitrary Signed Range [High, Low] 6-21
Example 6-27. Simplified Clipping to an Arbitrary Signed Range 6-21
Example 6-28. Clipping to an Arbitrary Unsigned Range [High, Low] 6-22
Example 6-29. Complex Multiply by a Constant 6-24
Example 6-30. Using PTEST to Separate Vectorizable and Non-Vectorizable Loop Iterations 6-25
Example 6-31. Using Variable BLEND to Vectorize Heterogeneous Loops 6-26
Example 6-32. Baseline C Code for Mandelbrot Set Map Evaluation 6-26
Example 6-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics 6-27
Example 6-34. A Large Load after a Series of Small Stores (Penalty) 6-29
Example 6-35. Accessing Data without Delay 6-29
Example 6-36. A Series of Small Loads after a Large Store 6-30
Example 6-37. Eliminating Delay for a Series of Small Loads after a Large Store 6-30
Example 6-38. Un-optimized Reverse Memory Copy in C 6-32
Example 6-39. Using PSHUFB to Reverse Byte Ordering 16 Bytes at a Time 6-33
Example 6-40. Table Look-up Operations in C Code 6-35
Example 6-41. Shift Techniques on Non-Vectorizable Table Look-up 6-36
Example 6-42. PEXTRD Techniques on Non-Vectorizable Table Look-up 6-37
Example 6-43. Pseudo-Code Flow of AES Counter Mode Operation 6-39
Example 6-44. AES128-CTR Implementation with Eight Block in Parallel 6-40
Example 6-45. AES128 Key Expansion 6-48
Example 6-46. Compress 32-bit Integers into 5-bit Buckets 6-51
Example 6-47. Decompression of a Stream of 5-bit Integers into 32-bit Elements 6-52
Example 7-1. Pseudocode for Horizontal (xyz, AoS) Computation 7-5
Example 7-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation 7-5
Example 7-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS 7-5
Example 7-4. Swizzling Data Using UNPCKxxx Instructions 7-6
Example 7-5. Deswizzling Single-Precision SIMD Data 7-7
Example 7-6. Deswizzling Data Using SIMD Integer Instructions 7-8
Example 7-7. Horizontal Add Using MOVHLPS/MOVLHPS 7-10
Example 7-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS 7-10
Example 7-9. Dot Product of Vector Length 4 Using SSE/SSE2 7-12
Example 7-10. Dot Product of Vector Length 4 Using SSE3 7-12
Example 7-11. Dot Product of Vector Length 4 Using SSE4.1 7-12
Example 7-12. Unrolled Implementation of Four Dot Products 7-13
Example 7-13. Normalization of an Array of Vectors 7-13
Example 7-14. Normalize (x, y, z) Components of an Array of Vectors Using SSE2 7-14
Example 7-15. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1 7-15
Example 7-16. Data Organization in Memory for AOS Vector-Matrix Multiplication 7-16
Example 7-17. AOS Vector-Matrix Multiplication with HADDPS 7-16
Example 7-18. AOS Vector-Matrix Multiplication with DPPS 7-17
Example 7-19. Data Organization in Memory for SOA Vector-Matrix Multiplication 7-18
Example 7-20. Vector-Matrix Multiplication with Native SOA Data Layout 7-19

EXAMPLES

PAGE

Example 8-1. VPDPBUSD Implementation 8-3
Example 8-2. Quantization of Activations (Scalar) 8-5
Example 8-3. Quantization of Activations (Vectorized) 8-5
Example 8-4. Direct Convolution 8-12
Example 8-5. Convolution for Layers with Low OFM Count 8-15
Example 8-6. Basic PostConv 8-16
Example 8-7. Uint8 Residual Input 8-18
Example 8-8. 8x8 Average Pooling with Stride 1 of 8x8 Layers 8-18
Example 8-9. Unfused Vectorized Pooling 8-19
Example 8-10. Caffe Scalar Code for Pixel Shuffler 8-20
Example 8-11. Computing Output Offset for Fused Pixel Shuffler 8-21
Example 8-12. Sigmoid Approximation with Minimax Polynomials 8-22
Example 8-13. Sigmoid Approximation with Scalef 8-24
Example 8-14. Pseudocode for Finding Top K 8-26
Example 9-1. Pseudo-code Using CLFLUSH 9-10
Example 9-2. Flushing Cache Lines Using CLFLUSH or CLFLUSHOPT 9-12
Example 9-3. Populating an Array for Circular Pointer Chasing with Constant Stride 9-13
Example 9-4. Prefetch Scheduling Distance 9-16
Example 9-5. Using Prefetch Concatenation 9-17
Example 9-6. Concatenation and Unrolling the Last Iteration of Inner Loop 9-17
Example 9-7. Data Access of a 3D Geometry Engine without Strip-mining 9-21
Example 9-8. Data Access of a 3D Geometry Engine with Strip-mining 9-22
Example 9-9. Using HW Prefetch to Improve Read-Once Memory Traffic 9-23
Example 9-10. Basic Algorithm of a Simple Memory Copy 9-27
Example 9-11. A Memory Copy Routine Using Software Prefetch 9-27
Example 9-12. Memory Copy Using Hardware Prefetch and Bus Segmentation 9-29
Example 10-1. Code Using libnuma to Find the Maximum Number of NUMA Node 10-2
Example 11-1. Serial Execution of Producer and Consumer Work Items 11-5
Example 11-2. Basic Structure of Implementing Producer Consumer Threads 11-6
Example 11-3. Thread Function for an Interlaced Producer Consumer Model 11-7
Example 11-4. Spin-wait Loop and PAUSE Instructions 11-12
Example 11-5. Coding Pitfall using Spin Wait Loop 11-15
Example 11-6. Placement of Synchronization and Regular Variables 11-16
Example 11-7. Declaring Synchronization Variables without Sharing a Cache Line 11-17
Example 11-8. Batched Implementation of the Producer Consumer Threads 11-21
Example 11-9. Parallel Memory Initialization Technique Using OpenMP and NUMA 11-25
Example 13-1. Compute 64-bit Quotient and Remainder with 64-bit Divisor 13-3
Example 13-2. Quotient and Remainder of 128-bit Dividend with 64-bit Divisor 13-4
Example 14-1. A Hash Function Examples 14-4
Example 14-2. Hash Function Using CRC32 14-5
Example 14-3. Strlen() Using General-Purpose Instructions 14-7
Example 14-4. Sub-optimal PCMPISTRI Implementation of EOS handling 14-9
Example 14-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency 14-9
Example 14-6. WordCnt() Using C and Byte-Scanning Technique 14-11
Example 14-7. WordCnt() Using PCMPISTRM 14-12
Example 14-8. KMP Substring Search in C 14-14
Example 14-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic 14-15
Example 14-10. Substring Search Using PCMPISTRI and KMP Overlap Table 14-17
Example 14-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic 14-19
Example 14-12. I Equivalent Strupr() Using PCMPISTRM Intrinsic 14-22
Example 14-13. UTF16 VerStrlen() Using C and Table Lookup Technique 14-23
Example 14-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI 14-24
Example 14-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI 14-26
Example 14-16. Replacement String Library Strcmp Using Intel® SSE4.2 14-27

EXAMPLES

PAGE

Example 14-17. High-level flow of Character Subset Validation for String Conversion 14-29
Example 14-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI 14-29
Example 14-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing 14-32
Example 14-20. Conversion of 64-bit Integer to ASCII 14-34
Example 14-21. Conversion of 64-bit Integer to ASCII without Integer Division 14-35
Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 14-37
Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 14-43
Example 14-24. MULX and Carry Chain in Large Integer Numeric 14-48
Example 14-25. Building-block Macro Used in Binary Decimal Floating-point Operations 14-48
Example 15-1. Cartesian Coordinate Transformation with Intrinsics 15-3
Example 15-2. Cartesian Coordinate Transformation with Assembly 15-5
Example 15-3. Direct Polynomial Calculation 15-7
Example 15-4. Function Calls and Intel® AVX/Intel® SSE transitions 15-11
Example 15-5. AoS to SoA Conversion of Complex Numbers in C Code 15-14
Example 15-6. Aos to SoA Conversion of Complex Numbers Using Intel® AVX 15-15
Example 15-7. Register Overlap Method for Median of 3 Numbers 15-17
Example 15-8. Data Gather - Intel® AVX versus Scalar Code 15-19
Example 15-9. Scatter Operation Using Intel® AVX 15-21
Example 15-10. SAXPY using Intel® AVX 15-22
Example 15-11. Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation 15-24
Example 15-12. SAXPY Implementations for Unaligned Data Addresses 15-24
Example 15-13. Loop with Conditional Expression 15-28
Example 15-14. Handling Loop Conditional with VMASKMOV 15-28
Example 15-15. Three-Tap Filter in C Code 15-29
Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD 15-29
Example 15-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS 15-30
Example 15-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code 15-31
Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends 15-33
Example 15-20. 8x8 Matrix Transpose Using VINSERTPS 15-36
Example 15-21. Port 5 versus Load Port Shuffles 15-38
Example 15-22. Divide Using DIVPS for 24-bit Accuracy 15-41
Example 15-23. Divide Using RCPPS 11-bit Approximation 15-41
Example 15-24. Divide Using RCPPS and Newton-Raphson Iteration 15-41
Example 15-25. Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy 15-42
Example 15-26. Reciprocal Square Root Using RSQRTPS 11-bit Approximation 15-43
Example 15-27. Reciprocal Square Root Using RSQRTPS and Newton-Raphson Iteration 15-43
Example 15-28. Square Root Using SQRTPS for 24-bit Accuracy 15-44
Example 15-29. Square Root Using RSQRTPS 11-bit Approximation 15-44
Example 15-30. Square Root Using RSQRTPS and One Taylor Series Expansion 15-45
Example 15-31. Array Sub Sums Algorithm 15-47
Example 15-32. Single-Precision to Half-Precision Conversion 15-48
Example 15-33. Half-Precision to Single-Precision Conversion 15-50
Example 15-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision 15-51
Example 15-35. FP Mul/FP Add Versus FMA 15-53
Example 15-36. Unrolling to Hide Dependent FP Add Latency 15-53
Example 15-37. FP Mul/FP Add Versus FMA 15-55
Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2 15-56
Example 15-39. Separable KLT Intra-block Transformation Using AVX2 15-58
Example 15-40. Macros for Parallel Moduli/Remainder Calculation 15-64
Example 15-41. Signed 64-bit Integer Conversion Utility 15-65
Example 15-42. Unsigned 63-bit Integer Conversion Utility 15-66
Example 15-43. Access Patterns Favoring Non-VGATHER Techniques 15-70
Example 15-44. Access Patterns Likely to Favor VGATHER Techniques 15-71
Example 15-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD 15-72

EXAMPLES

PAGE

Example 15-46. AOS to SOA Transformation Alternatives 15-74
Example 15-47. Non-Strided AOS to SOA 15-75
Example 15-48. Conversion to Throughput-Reduced MMX sequence to Intel® AVX2 Alternative 15-77
Example 16-1. Unoptimized Sleep Loop 16-14
Example 16-2. Power Consumption Friendly Sleep Loop Using PAUSE 16-14
Example 17-1. Cartesian Coordinate System Rotation with Intrinsics 17-4
Example 17-2. Cartesian Coordinate System Rotation with Assembly 17-6
Example 17-3. Masking with Intrinsics 17-10
Example 17-4. Masking with Assembly 17-11
Example 17-5. Masking Example 17-12
Example 17-6. Masking vs. Blending Example 1 17-13
Example 17-7. Masking vs. Blending Example 2 17-14
Example 17-8. Multiple Condition Execution 17-15
Example 17-9. Peeling and Remainder Masking 17-17
Example 17-10. Comparing Intel® AVX-512 Data Compress with Alternative 1 17-21
Example 17-11. Comparing Intel® AVX-512 Data Compress with Alternative 2 17-22
Example 17-12. Comparing Intel® AVX-512 Data Compress with Alternative 3 17-23
Example 17-13. Comparing Intel® AVX-512 Data Compress with Alternative 4 17-24
Example 17-14. Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives 17-26
Example 17-15. Comparing Ternary Logic to Alternative 1 17-28
Example 17-16. Comparing Ternary Logic to Alternatives 2 and 3 17-29
Example 17-17. Matrix Transpose Alternatives 17-33
Example 17-18. Broadcast Executed on Load Ports Alternatives 17-35
Example 17-19. 16-bit Broadcast Executed on Port 5 17-35
Example 17-20. Embedded vs Non-Embedded Rounding 17-36
Example 17-21. Scalar Scatter 17-39
Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics 17-41
Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly 17-42
Example 17-24. Scatter Implementation Alternatives 17-53
Example 17-25. Scalar vs. Vector Update Using AVX-512CD 17-55
Example 17-26. Improvement with VPERMB Implementation 17-57
Example 17-27. Improvement with VPERMI2B Implementation 17-60
Example 17-28. Improvement with VPMULTISHIFTQB Implementation 17-63
Example 17-29. 256-bit Code vs. 256-bit Code Mixed with 512-bit Code 17-66
Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture . 17-68
Example 17-31. Gather to Shuffle in Strided Loads Example 17-71
Example 17-32. Gather to Shuffle in Strided Stores Example 17-73
Example 17-33. Gather to Shuffle in Adjacent Loads Example 17-74
Example 17-34. Data Alignment 17-75
Example 17-35. Vectorized 32-bit Float Division, 24 Bits 17-82
Example 17-36. Vectorized 32-bit Float Division, 23 and 14 Bits 17-82
Example 17-37. Reciprocal Square Root, 22 Bits 17-82
Example 17-38. Reciprocal Square Root, 23 and 14 Bits 17-83
Example 17-39. Square Root , 24 Bits 17-83
Example 17-40. Square Root , 23 and 14 Bits 17-83
Example 17-41. Dividing Packed Doubles, 53 and 52 Bits 17-84
Example 17-42. Dividing Packed Doubles, 26 and 14 Bits 17-84
Example 17-43. Reciprocal Square Root of Doubles, 51 Bits 17-85
Example 17-44. Reciprocal Square Root of Doubles, 52 and 50 Bits 17-86
Example 17-45. Reciprocal Square Root of Doubles, 26 and 14 Bits 17-87
Example 17-46. Square Root of Packed Doubles, 53 and 52 Bits 17-87
Example 17-47. Square Root of Packed Doubles, 26 and 14 Bits 17-88
Example 18-1. Converting a Complex-Valued Mask to a Real-Valued Mask by Duplicating Adjacent Bits . 18-7
Example 18-2. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask By AND-Combining Ad-

EXAMPLES

PAGE

jacent Bits 18-8
Example 18-3. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask by OR-Combining Adjacent Bits 18-9
Example 18-4. Function to Implement the 16-Bit Compress Operation on FP16 Vector Elements 18-17
Example 18-5. Function that Performs Fast Floating-Point Minimum Using Integer Instructions 18-19
Example 19-1. Legacy Intel® AES-NI vs. Vector AES 19-1
Example 19-2. SM4 GFNI Encryption Round Example 19-3
Example 20-1. Pseudo-Code for the TILEZERO, TILELOAD, and TILESTORE Instructions 20-6
Example 20-2. B Matrix Re-Layout Procedure 20-6
Example 20-3. Original Layout of 32x16 bfloat16 B-Matrix 20-7
Example 20-4. Re-Layout of 32x16 bfloat16 B-Matrix 20-7
Example 20-5. Original Layout of 64 x 16 unt8 B-Matrix 20-8
Example 20-6. Re-Layout of 64x16 bfloat16 B-Matrix 20-8
Example 20-7. Common Defines 20-9
Example 20-8. Reference GEMM Implementation 20-10
Example 20-9. K-Dimension Loop as Innermost Loop–A, a Highly Inefficient Approach 20-11
Example 20-10. Innermost Loop Tile Pre-Loading 20-12
Example 20-11. Switched Order of M_ACC and N_ACC Loops 20-12
Example 20-12. Optimized GEMM Implementation 20-13
Example 20-13. Dimension of Matrices, Data Types, and Tile Sizes 20-14
Example 20-14. Optimized GEMM Assembly Language Implementation 20-14
Example 20-15. Activations Layout Procedure 20-15
Example 20-16. Weights Re-Layout Procedure 20-16
Example 20-17. Common Defines for Convolution 20-19
Example 20-18. Optimized Direct Convolution Implementation 20-20
Example 20-19. Additional Defines for Convolution with Cache Blocking 20-21
Example 20-20. Optimized Convolution Implementation with Cache Blocking 20-22
Example 20-21. Convolution Code Fused with Post-Convolution Code 20-30
Example 20-22. An Example of a Short GEMM Fused and Pipelined with Quantization and ReLU 20-33
Example 20-23. Two Blocks of 16 Cache Lines of 32-bit Floats Converted to One Block of 16 Cache Lines of 16-bit BFloat 20-37
Example 20-24. Using Unsigned Saturation 20-37
Example 20-25. Prefetching Rows to the DCU 20-41
Example 20-26. BF16 Matrix Transpose (32x8 to 8x32) 20-42
Example 20-27. BF16 VNNI-to-VNNI Transpose (8x8 to 2x32) 20-46
Example 20-28. BF16 Flat-to-VNNI Transpose (16x8 to 4x32) 20-49
Example 20-29. BF16 Flat-to-VNNI Re-Layout 20-51
Example 20-30. GEMM Parallelized with omp Parallel for Collapse 20-55
Example 20-31. Byte Decompression Code with Intel® AVX-512 Intrinsics 20-58
Example 20-32. Identification of Tile Shape Using Parameter m, n, k 20-60
Example 20-33. Intel® AMX Intrinsics Header File 20-60
Example 20-34. Intel® AMX Intrinsics Usage 20-63
Example 20-35. Compiler-Generated Assembly-Level Code from Example 20-30 20-64
Example 20-36. Compiler-Generated Assembly-Level Code Where Tile Register Save/Restore is Optimized Away 20-65

CHAPTER 1
INTRODUCTION

1.1 ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors.

The target audience for this manual includes software programmers and compiler writers. This manual assumes that the reader is familiar with the basics of the IA-32 architecture and has access to the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. A detailed understanding of Intel 64 and IA-32 processors is often required. In many cases, knowledge of the underlying microarchitectures is required.

The design guidelines discussed in this manual for developing high-performance software generally apply to current and future IA-32 and Intel 64 processors. In most cases, coding rules apply to software running in 64-bit mode of Intel 64 architecture, compatibility mode of Intel 64 architecture, and IA-32 modes (IA-32 modes are supported in IA-32 and Intel 64 architectures). Coding rules specific to 64-bit modes are noted separately.

NOTE

A public repository is available with open source code samples from select chapters of this manual. These code samples are released under a 0-Clause BSD license. Intel provides additional code samples and updates to the repository as the samples are created and verified.

Public repository: https://github.com/intel/optimization-manual.

Link to license: https://github.com/intel/optimization-manual/blob/master/COPYING.

1.2 TUNING YOUR APPLICATION

Tuning an application for high performance on any Intel 64 or IA-32 processor requires understanding and basic skills in:

- Intel 64 and IA-32 architecture.
- C and Assembly language.
- Hot-spot regions in the application that impact performance.
- Optimization capabilities of the compiler.
- Techniques used to evaluate application performance.

The Intel® VTune™ Profiler can help you analyze and locate hot-spot regions in your applications.

On many Intel processors, this tool can monitor an application through a selection of performance monitoring events and analyze the performance event data that is gathered during code execution.

This manual also describes data that can be gathered using the performance counters through the processor's performance monitoring events.

1.3 INTEL PROCESSORS SUPPORTING THE INTEL® 64 ARCHITECTURE

The following is a list of Intel processors, series, and product families that support the Intel 64 Architecture[1]. The list is organized by microarchitecture.

Table 1-1. Intel Processors Organized by Microarchitecture

Microarchitecture	Processor(s), Series, Product(s)
Nehalem microarchitecture (45nm)	• The Intel® Core™ i7 processor • Intel® Xeon® processor 3400, 5500, 7500 series
Westmere microarchitecture (32nm)	• Intel® Xeon® processor 5600 series • Intel® Xeon® processor E7 • Various Intel® Core™ i7, i5, i3 processors
Sandy Bridge microarchitecture	• Intel® Xeon® processor E5 family • Intel® Xeon® processor E3-1200 family • Intel® Xeon® processor E7-8800/4800/2800 product families • Intel® Core™ i7-3930K processor • 2nd generation Intel® Core™ i7-2xxx processor series • Intel® Core™ i3-2xxx processor series
Ivy Bridge microarchitecture	• Intel® Xeon® processor E7-8800/4800/2800 v2 product families • Intel® Xeon® processor E3-1200 v2 product family • 3rd generation Intel® Core™ processors
Ivy Bridge-E microarchitecture	• Intel® Xeon® processor E5-4600/2600/1600 v2 product families • Intel® Xeon® processor E5-2400/1400 v2 product families • Intel® Core™ i7-49xx Processor Extreme Edition
Haswell microarchitecture	• Intel® Xeon® processor E3-1200 v3 product family • 4th Generation Intel® Core™ processors
Haswell-E microarchitecture	• Intel® Xeon® processor E5-2600/1600 v3 product families • Intel® Core™ i7-59xx Processor Extreme Edition
Airmont microarchitecture	Intel® Atom® processor Z8000 series
Silvermont microarchitecture	Intel® Atom® processor Z3400 series
Broadwell microarchitecture	• Intel® Core™ M processor family • 5th generation Intel® Core™ processors • Intel® Xeon® processor D-1500 product family • Intel® Xeon® processor E5 v4 family
Skylake microarchitecture	• Intel® Xeon® Scalable processor family • Intel® Xeon® processor E3-1500m v5 product family • 6th generation Intel® Core™ processors
Kaby Lake microarchitecture	7th generation Intel® Core™ processors

1. For more about this architecture, visit: https://www.intel.com/content/www/us/en/architecture-and-technology/microarchitecture/intel-64-architecture-general.html

Table 1-1. (Contd.)Intel Processors Organized by Microarchitecture

Microarchitecture	Processor(s), Series, Product(s)
Goldmont microarchitecture	• Intel Atom® processor C series • Intel Atom® processor X series • Intel® Pentium® processor J series • Intel® Celeron® processor J series • Intel® Celeron® processor N series
Knights Landing microarchitecture	Intel® Xeon Phi™ Processor 3200, 5200, 7200 series
Goldmont Plus microarchitecture	• Intel® Pentium® Silver processor series • Intel® Celeron® processor J series • Intel® Celeron® processor N series
Coffee Lake microarchitecture	• Intel® Xeon® E processors • 8th generation Intel® Core™ processors • 9th generation Intel® Core™ processors
Knights Mill microarchitecture	Intel® Xeon® Phi™ Processor 7215, 7285, 7295 Series
Cascade Lake microarchitecture	2nd generation Intel® Xeon® Scalable processor family
Ice Lake microarchitecture	• Some of the 3rd generation Intel® Xeon® Scalable processor family • Some 10th generation Intel® Core™ processors
Comet Lake microarchitecture	Some 10th generation Intel® Core™ processors
Tiger Lake microarchitecture	Some 11th generation Intel® Core™ processors
Rocket Lake microarchitecture	Some 11th generation Intel® Core™ processors
Cooper Lake microarchitecture	Some of the 3rd generation Intel® Xeon® Scalable processor family
Gracemont microarchitecture	Intel N-Series processors
Alder Lake microarchitecture	12th generation Intel® Core™ processors
Raptor Lake microarchitecture	• 13th generation Intel® Core™ processors • 14th generation Intel® Core™ processors
Sapphire Rapids microarchitecture	4th generation Intel® Xeon® Scalable processor family
Emerald Rapids microarchitecture	5th generation Intel® Xeon® Scalable processor family
Granite Lake microarchitecture	6th generation Intel® Xeon® Scalable processor family
Meteor Lake microarchitecture	1st generation Intel® Core™ Ultra processors

1.4 ORGANIZATION OF THIS MANUAL

This manual is divided into two volumes. The first considers the optimization of newer products and technologies. Volume Two considers older technology that may not currently be supported.

1.4.1 CHAPTER SUMMARIES

1.4.1.1 Volume 1

- **Chapter 1: Introduction:** Defines the purpose and outlines the contents of this manual.
- **Chapter 2: Intel® 64 and IA-32 Processor Architectures:** Describes the microarchitecture of recent Intel 64 and IA-32 processor families, and other features relevant to software optimization.
- **Chapter 3: General Optimization Guidelines:** Describes general code development and optimization techniques that apply to all applications designed to take advantage of the common features of current Intel processors.
- **Chapter 4: Intel Atom® Processor Architecture:** Describes the microarchitecture of recent Intel Atom processor families, and other features relevant to software optimization.
- **Chapter 5: Coding for SIMD Architectures:** Describes techniques and concepts for using the SIMD integer and SIMD floating-point instructions provided by the MMX™ technology, Streaming SIMD Extensions, Streaming SIMD Extensions 2, Streaming SIMD Extensions 3, SSSE3, and SSE4.1.
- **Chapter 6: Optimizing for SIMD Integer Applications**: Provides optimization suggestions and common building blocks for applications that use the 128-bit SIMD integer instructions.
- **Chapter 7: Optimizing for SIMD Floating-point Applications:** Provides optimization suggestions and common building blocks for applications that use the single-precision and double-precision SIMD floating-point instructions.
- **Chapter 8: INT8 Deep Learning Inference:** Describes INT8 as a data type for Deep learning Inference on Intel technology. The document covers both AVX-512 implementations and implementations using the new Intel® DL Boost Instructions.
- **Chapter 9: Optimizing Cache Usage**: Describes how to use the PREFETCH instruction, cache control management instructions to optimize cache usage, and the deterministic cache parameters.
- **Chapter 10: Introducing Sub-NUMA Clustering:** Describes Sub-NUMA Clustering (SNC), a mode for improving average latency from last level cache (LLC) to local memory.
- **Chapter 11: Multicore and Intel® Hyper-Threading Technology:** Describes guidelines and techniques for optimizing multithreaded applications to achieve optimal performance scaling. Use these when targeting multicore processor, processors supporting Hyper-Threading Technology, or multiprocessor (MP) systems.
- **Chapter 12: Intel® Optane™ DC Persistent Memory:** Provides optimization suggestions for applications that use Intel® Optane™ DC Persistent Memory.
- **Chapter 13: 64-Bit Mode Coding Guidelines:** This chapter describes a set of additional coding guidelines for application software written to run in 64-bit mode.
- **Chapter 14: SSE4.2 and SIMD Programming for Text-Processing/Lexing/Parsing:** Describes SIMD techniques of using SSE4.2 along with other instruction extensions to improve text/string processing and lexing/parsing applications.
- **Chapter 15: Optimizations for Intel® AVX, FMA, and Intel® AVX2:** Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions, FMA, and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- **Chapter 16: Intel Transactional Synchronization Extensions:** Tuning recommendations to use lock elision techniques with Intel Transactional Synchronization Extensions to optimize multi-threaded software with contended locks.
- **Chapter 16: Power Optimization for Mobile Usages:** This chapter provides background on power saving techniques in mobile processors and makes recommendations that developers can leverage to provide longer battery life.
- **Chapter 17: Software Optimization for Intel® AVX-512 Instructions:** Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions 512.
- **Chapter 18: Intel® Advanced Vector Extensions 512-FP16 Instruction Set for Intel® Xeon® Processors: D**escribes the addition of the FP16 ISA for Intel AVX-512 to handle IEEE 754-2019 compliant half-precision floating-point operations.
- **Chapter 19: Cryptography & Finite Field Arithmetic Enhancements:** Describes the new instruction extensions designated for acceleration of cryptography flows and finite field arithmetic.
- **Chapter 20: Intel® Advanced Matrix Extensions (Intel® AMX): Describes best practices to** optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP microarchitecture. It extends the public documentation on Optimizing DL code with DL Boost instructions.
- **Chapter 21: Intel® QuickAssist Technology (Intel® QAT):** Describes software development guidelines for the Intel® QuickAssist Technology (Intel® QAT) API. This API supports both the Cryptographic and Data Compression services.
- **Appendix A: Application Performance Tools:** Introduces tools for analyzing and enhancing application performance without having to write assembly code.
- **Appendix B: Using Performance Monitoring Events:** Provides information on the Top-Down Analysis Method and information on how to use performance events specific to the Intel Xeon processor 5500 series, processors based on Sandy Bridge microarchitecture, and Intel Core Solo and Intel Core Duo processors.
- **Appendix C: Intel Architecture Optimization with Large Code Pages:** Provides information on how the performance of runtimes can be improved by using large code pages.

1.4.1.2 Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures

- **Chapter 1: Haswell Microarchitecture:** Describes the Haswell microarchitecture.
- **Chapter 2: Sandy Bridge Microarchitecture:** Describes the Sandy Bridge microarchitecture and associated considerations.
- **Chapter 3: Intel® Core™ Microarchitecture and Enhanced Intel® Core™ Microarchitecture:** Describes the Intel® Core™ and Enhanced Intel® Core ™microarchitectures and associated considerations.
- **Chapter 4: Nehalem Microarchitecture:** Describes the Sandy Bridge microarchitecture and associated considerations.
- **Chapter 5: Knights Landing Microarchitecture Optimization:** Describes the Sandy Bridge microarchitecture and associated considerations, including Multithreading and Intel® HyperThreading Technology (Intel® HT).
- **Chapter 6: Earlier Generations of Intel Atom® Microarchitecture and Software Optimization:** Describes the microarchitecture of earlier generations of processor families based on Intel Atom microarchitecture, and software optimization techniques targeting Intel Atom microarchitecture.

1.5 GLOSSARY

Table 1-2 provides definitions of commonly used terms throughout this volume.

Table 1-2. Term Definitions

Term	Description
Arithmetic formats	Sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values.
CFP16	Complex-valued floating-point format comprising two FP16 values representing the real and imaginary values respectively. When used in SIMD, the individual real/imaginary values from each complex value are interleaved in the register.
Denormal	A subset of denormalized numbers that fill the underflow gap around zero in floating-point arithmetic.
FP16	Half precision 16-bit floating-point data format.
FP32	Single precision 32-bit floating-point data format.
FP64	Double precision 64-bit floating-point data format.
FFT	Fast Fourier Transform.
IEEE 754-2019	The current IEEE Standard for Floating-Point Arithmetic used in Intel® AVX-512 FP16 instructions.
Intel® AVX	Intel® Advanced Vector Extensions.
Intel® AVX2	Intel® Advanced Vector Extensions 2.
Intel® AVX-512	Intel® Advanced Vector Extensions 512.
Intel® AVX-512 FP16	ISA for handling half precision floating-point, added as an extension to Intel AVX-512.
Intrinsic	A function that can be called from a high-level language, like C/C++, which gives direct access to the underlying ISA. Intrinsics allow the programmer to bypass the compiler and directly specify that a particular instruction be used.
ISA	Instruction Set Architecture[1]: a part of the abstract model of a computer, which generally defines how software controls the CPU[2].
MMSE	Minimum Mean Squared Error.
NaN	Not A Number. A way to represent a value that is undefined or unrepresentable. For example, the square root of a negative number would generate a NaN value.
Normal	A floating-point number that can be represented without leading zeros in its significand.
SIMD	Single instruction, multiple data. A way of packing several data elements into a single container and operating on them all at once.[3]
SINR	Signal-to-Interference-plus-Noise Ratio.
SSE	SIMD Streaming Extensions[4].

Table 1-2. Term Definitions

Term	Description
µop	Also uop. Refers to **micro-operations** (micro-ops) and is usually identified in code as UOP. Micro-operations are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes referred to as macro-instructions in this context).[5]

NOTES:

1. See Intel's Instruction Set Architecture landing page.
2. See Wikipedia.
3. See Chapter 5, "Coding for SIMD Architectures"
4. See Intel's Instruction Set Extensions Technology Support landing page.
5. See Wikipedia for a deep dive.

1.6 RELATED INFORMATION

For more information on the Intel® architecture, techniques, and the processor architecture terminology, the following are of particular interest. Intel publishes new and updated content continuously.

Table 1-3. Additional References in and Beyond this Document

Title	Description
Intel® 64 and IA-32 Architectures Software Developer's Manual	These manuals describe the architecture and programming environment of the Intel® 64 and IA-32 architectures. This links directly to the PDF containing all 4 columns of the content.
CPU Tuning and Optimization Guides	This page includes guides covering specific Processors and technologies.
Intel® 64 Architecture Processor Topology Enumeration	Covers the topology enumeration algorithm for single-socket to multiple-socket platforms using Intel® 64 and IA-32 processors.
Intel® Artificial Intelligence (Intel® AI) Solutions landing page	The official source for development using Intel® AI solutions supporting Deep Learning (DL) and Machine Learning (ML). Includes a section with documentation.
Support for Intel® Processors	Landing page for support information for Intel® processors including featured content, downloads, specifications, warranty, and community posts.
Get Started with Intel® Fortran Compiler Classic and Intel® Fortran Compiler	A guide to the basics of using Intel® Fortran Compilers: ifort and ifx. Please note: IFORT will be discontinued in October 2024.
Intel® C++ Compiler Classic (ICC) Developer Guide and Reference	Contains information about the Intel® C++ Compiler Classic (icc for Linux* and icl for Windows*) compiler and runtime environment.
Intel® Data Streaming Accelerator User Guide	

Table 1-3. Additional References in and Beyond this Document

Title	Description
Intel® Developer Zone Landing Page	The official source for developing on Intel® hardware and software. Includes documentation.
Intel® Developer Catalog	Find software and tools to develop and deploy solutions optimized for Intel® architecture.
Intel® Development Topics & Technologies landing page	A landing page devoted to everything from storage to computer vision (CV).
Intel® Distribution of OpenVino™ Toolkit landing page	The official source for the Intel® distribution of OpenVINO™, an open source toolkit that simplifies deployment. Includes a section with documentation.
Intel® Hyper-Threading Technology (Intel® HT Technology)	An overview of Intel® HT Technology. This links directly to the PDF.
Intel® In-Memory Analytics Accelerator Architecture Specification	Describes the architecture of the Intel® In-Memory Analytics Accelerator (Intel® IAA). This links directly to the PDF.
Intel® Instruction Set Extensions Technology Support	A landing page dedicated to all content related to supporting the Intel® ISE technologies. Includes the Intel® SSE4 Programming Reference.
Intel® oneAPI Data Analytics Library Landing Page	The official source for development using Intel® one API Data Analytics Library (oneDAL). Includes a section with documentation.
Intel® oneAPI DPD++/C++ Compiler	Intel® oneAPI DPC++/C++ Compile, a standards-based, cross-architecture compiler and update to both ifort and ifx.
Intel® QuickAssist Technology (Intel® QAT)	The official source for the Intel® QuickAssist Technology (Intel® QAT). Includes a section with documentation.
Intel® VTune™ Profiler User Guide	A comprehensive overview of the product functionality, tuning methodologies, workflows, and instructions to use the Intel® VTune™ Profiler performance analysis tool. This links directly to the PDF.
Intel® Xeon® Processors Technical Resources Page	A landing page including technical resources for all Intel® Xeon® Scalable processors.
C2C - False Sharing Detection in Linux Perf	An introduction to perf c2c in Linux.
Developing Multi-Threaded Applications: A Platform Consistent Approach	The objective of the Multithreading Consistency Guide is to provide guidelines for developing efficient multithreaded applications across Intel-based symmetric multiprocessors (SMP) and/or systems with Intel® Hyper-Threading Technology (Intel® HT). (2005)

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter overviews features relevant to software optimization for current generations of Intel® 64 and IA-32 processors[1]. These features include:

- Microarchitectures that enable executing instructions with high throughput at high clock speeds, a high-speed cache hierarchy, and a high-speed system bus.
- Intel® Hyper-Threading Technology[2] (Intel® HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- Single Instruction Multiple Data (SIMD) instruction extensions: MMX™ technology, Streaming SIMD Extensions (Intel® SSE), Streaming SIMD Extensions 2 (Intel® SSE2), Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® SSE4.1, and Intel® SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
- Intel® Thread Director.

2.1 6TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY

2.1.1 THE REDWOOD COVE MICROARCHITECTURE

The Redwood Cove microarchitecture is the successor of the Golden Cove microarchitecture. The Redwood Cove microarchitecture introduces the following enhancements:

- Improvements for larger code footprint workloads.
- Larger instruction cache: 32K→64K.
- Branch Hint x86 architecture extension.
- Code Software Prefetch x86 architecture extension (Granite Rapids only).
- Improved LSD coverage: the IDQ can hold 192 µops per logical processor in single-thread mode or 96 µops per thread when SMT is active.
- Improved branch prediction and reduced average branch misprediction recovery latency.
- New LD+OP and MOV+OP macro fusions.

---

1. Intel® Atom® processors are covered in Chapter 4, "Intel Atom® Processor Architectures."
2. Intel® HT Technology requires a computer system with an Intel processor supporting hyper-threading and an Intel® HT Technology-enabled chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.

- EXE: 3-cycle Floating Point multiplication.
- Improved SMT performance and efficiency.
- Improved load lock performance.
- New HW data prefetcher to recognize and prefetch the “Array of Pointers” pattern.
- Mid-level-cache size increased to 2MBs for Client.
- Improved Memory Bandwidth.
  - Increased number of outstanding misses (48→64 Deeper MLC miss queues).
  - LLC Page Prefetcher.
- AMX supports FP16 for AI/ML (Granite Rapids only).

2.1.1.1 Branch Hint

If the branch predictor has stored information about a conditional branch, the predictor should use this information to predict the branch. If the predictor does not have stored information, the predictor predicts the branch to be **not-taken**. This is usually, but not always, correct.

Starting with the Redwood Cove microarchitecture, if the predictor has no stored information about a branch, the branch has the Intel® SSE2 branch taken hint (i.e., instruction prefix 3EH),

When the codec decodes the branch, it flips the branch’s prediction from not-taken to taken. It then flushes the pipeline in front of it and steers this pipeline to fetch the **taken path** of the branch.

If the branch is taken when executed, the hint reduces the misprediction penalty by replacing an execution time pipeline flush and re-steer with a decode time of one. Alternatively, if the branch is not-taken, the hint increases the branch misprediction penalty by introducing a new decode time and a new execution time flush and re-steer. Consequently, the hint should not be added to a **not-taken branch** since this increases the program’s run time (see Section 2.1.1.3).

The hint is only used when the predictor does not have stored information about the branch. To avoid code bloat and reducing the instruction fetch bandwidth, don’t add the hint to a branch in hot code—for example, a branch inside a loop with a high iteration count—because the predictor will likely have stored information about that branch. Ideally, the hint should only be added to infrequently executed branches that are mostly taken, but identifying those branches may be difficult. Compilers are advised to add the hints as part of profile-guided optimization, where the one-sided execution path cannot be laid out as a fall-through. The Redwood Cove microarchitecture introduces new performance monitoring events to guide hint placement (see Section 2.1.1.3).

Judiciously adding hints can reduce the program’s run time by reducing the branch misprediction penalty. Hints are especially useful for large code footprint workloads where the number of branches exceeds the predictor's capacity.

Details include:

- The hint only applies to JCCs, not CXZ and LOOP/LOOPCC[1].
- The microarchitecture is obliged to decode the Intel® SSE2 branch not-taken hint (i.e., instruction prefix 2EH), but otherwise ignores it.
  - There’s no need to hint that a branch is not-taken since, by default, it’s predicted not-taken if the predictor doesn’t have stored information about it.
- An instruction has the taken hint if it has at least one instruction prefix 3EH.
  - **Example:** If an instruction has the 3EH and 2EH prefixes, regardless of order, it still has the taken the hint.

1. As specified in section 2.1.1 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A Chapter 2, “Instruction Format”.

2.1.1.2 Profiling Support for Branch Hints

The following performance monitoring events may be used to identify branches that may benefit from adding the previous hint:

- FRONTEND_RETIRED.MISP_ANT (Event Code = C6H, Unit Mask = 02H): Always Not-Taken conditional branch instructions retired that were miss-predicted.
- FRONTEND_RETIRED.ANY_ANT (Event Code = C6H, Unit Mask = 03H): Always Not-Taken conditional branch instructions retired.
- BR_INST_RETIRED.COND (Event Code = C4H, Unit Mask = 11H): conditional branch instructions retired.

Note that collecting Last Branch Records (LBR) when sampling on the last two events is instrumental to determining the polarity of the branch (e.g., was the branch mostly taken or mostly not taken).

2.1.1.3 New Redwood Cove Macro-Fusions

The Redwood Cove microarchitecture supports two new types of instruction macro-fusion: MOV-OP and LOAD-OP. The new macro-fusions are only supported out of the µop-Cache (not supported on the legacy decode pipeline).

With a new MOV+OP macro-fusion, a register MOV instruction can be fused with following an OP instruction to form the non-destructive source operation form: c ← *a* op *b*, preserving both source operands.

For example:

Table 2-1. MOV+OP Macro-Fusion Example

Class	Example	Macro-Fused to:
MOV+OP	mov rax, rbx sub rax, rcx	rax = sub rbx, rcx

The two instructions are macro-fused to a single micro-operation cached in the µop-Cache. The macro-fused operation takes a single slot in IDQ and during allocation, execution, and retirement.

Similarly, with a new LOAD+OP macro-fusion, a LOAD instruction could be macro-fused by following an OP instruction to form a fused load-op micro-operation, which often cannot be encoded within the x86 instruction set.

For example:

Table 2-2. LD+OP Macro-Fusion Example

Class	Example	Macro-Fused to:
LD+OP	mov eax, [mem] sub eax, ebx	eax = sub [mem], ebx[1]

NOTES:

1. Note that only sub reg, [mem] is encodable within x86 instruction set.

Limitations of LOAD+OP Macro-Fusion

The following instruction pair can be macro-fused to the single DST:= OP [mem], SRC_REG micro-operation.

(1) MOV DST, [mem]

(2)OP DST, SRC_REG // SRC_REG cannot be immediate and must be different from DST

- LOAD and OP operations must be the same Operand Size 32/64 matching load data size.
- Load cannot be encoded using RIP-Relative form or have Index operand.
- Only integer and packed SIMD/floating point 128-bit vector operations are supported.
- The LOAD must be:
  - An integer load operation without sign-extension (opcode 0x8B).
  - SSE packed load instructions (movaps/pd, movups/pd, movdqa/u).
- No VEX or EVEX forms are supported.

2.1.1.4 Improved Memory BW

The Redwood Cove microarchitecture improves the peak memory bandwidth that a core can consume over the Golden Cove microarchitecture. This significantly increases the depth of the MLC miss queue (48→64 outstanding MLC misses) and introduces new hardware prefetcher algorithms.

2.1.1.5 Array of Pointers Prefetcher

An array of pointers is an array whose elements are pointers. The data associated with each element of the array is usually dynamically allocated elsewhere and can be accessed by dereferencing the pointer pointing to it. For example:

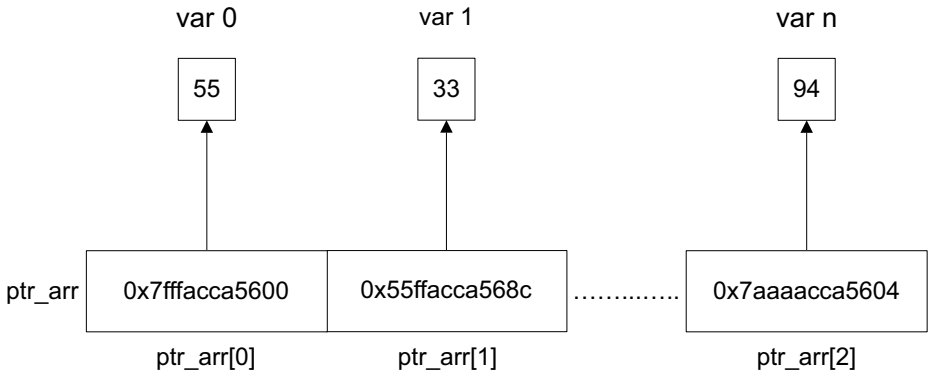


Figure 2-1. Layout of Array of Pointers Prefetchers

The Redwood Cove microarchitecture adds a new Array of Pointers (AOP) hardware prefetcher for detecting and prefetching an array of pointers references into the cache hierarchy. The AOP prefetcher treats the data prefetched for a constant stride load as a pointer and may issue prefetch requests to the memory addresses corresponding to the pointer's value.

Successful detection of the pattern allows AOP Prefetch to bring data to be accessed by the program in advance, saving costly cache misses.

To benefit most from the AOP Prefetching capabilities, we recommend the following:

The array of pointers pattern, in a nutshell, may look like this:

```
for (i=0; i<array_length; i++) {
(1)mov rax, array[i]
(2)mov rbx, [rax + offset]; data of rax used as a pointer
}
```

In the Redwood Cove microarchitecture, to benefit from AOP Prefetch, software must ensure that:

- 1st load operation is a regular integer load, naturally aligned to its data size (64-bit or 32-bit in legacy or compatibility mode)
- 2nd load operation depends upon the first load through the base operand and has no index operand.
  - The AOP Prefetcher does not track and would issue a prefetch ignoring the offset. Therefore, it is recommended to use small offsets or avoid using offsets at all.
  - The AOP Prefetcher avoids triggering prefetch if the offset exceeds 32 bytes.

Software that observes any issue with the AOP Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can deactivate it through bit 7 of MSR 0x1A4 (MSR_PREFETCH_CTRL). The AOP Prefetch is a data-dependent prefetcher. Security implications of data-dependent prefetching are discussed separately at:

https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html

2.1.1.6 LLC Page Prefetcher (LLCPP)

The LLC Page Prefetcher extends the Next-Page-Prefetcher algorithm, which detects a sequence of accesses that are about to reach the end of a page (either going up towards the next page in memory or down towards the previous page in memory) and triggers a linear prefetch to the start of the next predicted page. NPP's intends to prefetch the translation of the next page and initiate prefetching of the first few cache lines from that predicted page.

The LLC Page Prefetch enhances Next-Page Prefetcher in two significant ways:

- It issues prefetches two pages ahead in linear address space.
- The prefetch attempts to bring the entire 4Kb page to the last level cache using opportunistic IDI request slots without the DCU or MLC buffers.

LLCPP is effective when streaming through memory over a buffer greater than 8KB. Smaller buffer access might result in redundant prefetching to the last level cache.

Software that observes any issue with the LLC Page Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can disable LLC Page Prefetcher through bit 6 of MSR 0x1A4 (MSR_PREFETCH_CTRL).

2.2 THE SAPPHIRE RAPIDS MICROARCHITECTURE

Intel processors based on Sapphire Rapids microarchitecture use Golden Cove cores and support the following additional features:

- Intel® Advanced Matrix Extensions (Intel® AMX) (Chapter 20).
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) (Chapter 18).
- Intel® Data Streaming Accelerator (Intel® DSA)[1].

1. Please see the Intel® DSA Specification and Intel® DSA User Guide.

- Intel® In-Memory Analytics Accelerator (Intel® IAA)[1].
- Intel® Quick Assist Technology (Intel® QAT)(Chapter 21)

2.2.1 4TH GENERATION INTEL® XEON® SCALABLE FAMILY OF PROCESSORS

Intel's fourth generation Xeon® Scalable Family of Processors changes from a single-die monolithic design to multi-die Tiles.

- The server products are scalable from dual-socket to eight-socket configurations.
- The I/O is increased with PCI Express 5.0, DDR5 memory, and Compute Express Link 1.1.
- Packaging includes a multi-die chip with up to 4 tiles. Each tile is a 400mm2 SoC, providing compute cores and I/O.
- Each tile contains 15 Golden Cove cores (see Section 2.4). Its memory controller provides two channels of DDR5, up to eight channels across four tiles, and 28 PCIe 5.0 lanes, up to a maximum of 112 across four tiles.

2.3 THE ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE

The Alder Lake performance hybrid architecture combines two Intel architectures, combining the Golden Cove performant cores and the Gracemont efficient Atom cores onto a single SoC. For details on the Golden Cove microarchitecture, see Section 2.4. For details on the Crestmont microarchitecture, see Section 4.1

2.3.1 12TH GENERATION INTEL® CORE™ PROCESSORS SUPPORTING PERFORMANCE HYBRID ARCHITECTURE

12th Generation Intel® Core™ processors supporting performance hybrid architecture compriseup to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MBs Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has a symmetrical ISA and comes in a variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. These processors' P-cores can also have Intel Hyper-Threading Technology enabled. When the operating system (OS) decides to schedule all processors, all cores can be active simultaneously.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-cores and E-cores. To maintain symmetric ISA, the E-cores do not support the following features: Intel AVX-512, Intel AVX-512 FP-16, and Intel® TSX. The E-cores do support Intel AVX2 and Intel AVX-VNNI.

2.3.2 HYBRID SCHEDULING

2.3.2.1 Intel® Thread Director

Intel® Thread Director monitors software in real-time, giving hints to the operating system's scheduler, allowing it to make more intelligent and data-driven decisions on thread scheduling. With Intel Thread Director, the hardware provides runtime feedback to the OS per thread based on various IPC performance characteristics in the form of:

1. Please see the Intel® IAA Specification.

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director was first introduced in desktop and mobile variants of the 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.

A processor containing P-cores and E-cores with different performance characteristics challenges the operating system's scheduler. Additionally, different software threads see different performance ratios between the P-cores and E-cores. For example, the performance ratio between the P-cores and E-cores for highly vectorized floating-point code is higher than that for scalar integer code. So, when the operating system must make an optimal scheduling decision, it must be aware of the characteristics of the software threads that are candidates for scheduling. Suppose there are insufficient P-cores and a mix of software threads with different characteristics. In that case, the operating system should schedule those threads that benefit most from the P-cores onto those cores and schedule the others on the E-cores.

Intel Thread Director provides the necessary hint to the operating system about the characteristics of the software thread executing on each of the logical processors. The hint is dynamic and reflects the recent characteristics of the thread, i.e., it may change over time based on the dynamic instruction mix of the thread. The processor also considers microarchitecture factors to define the dynamic software thread characteristics.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the operating system via writing to configuration MSRs. The Intel Thread Director implementation on processors based on Alder Lake performance hybrid architecture defines four thread classes:

1. Non-vectorized integer or floating-point code.
2. Integer or floating-point vectorized code, excluding Intel® Deep Learning Boost (Intel® DL Boost) code.
3. Intel DL Boost code.
4. Pause (spin-wait) dominated code.

The dynamic code need not be 100% of the class definition. It should be large enough to be considered belonging to that class. Also, dynamic microarchitectural metrics such as consumed memory bandwidth or cache bandwidth may move software threads between classes. Example pseudo-code sequences for the Intel Thread Director classes available on processors based on Alder Lake performance hybrid architecture are provided in Examples 2-1 through 2-4.

Intel Thread Director also provides a table in system memory, only accessible to the operating system, that defines the P-core vs. E-core performance ratio per class. This allows the operating system to pick and choose the correct software thread for the correct logical processor.

In addition to the performance ratio between P-cores and E-cores, Intel Thread Director provides the energy efficiency ratio between those cores. The operating system can use this information when it prefers energy savings over maximum performance. For example, a background task such as indexing can be scheduled on the most energy-efficient core since its performance is less critical.

Example 2-1. Class 0 Pseudo-code Snippet

```
while (1)
{
    asm("xor rax, rax;"
        "add rax, 5;"
        "inc rax;"
    );
}
```

Example 2-2. Class 1 Pseudo-code Snippet

```
while (1)
{
    asm("vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"

        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
    );
}
```

Example 2-3. Class 2 Pseudo-code Snippet

```
while (1)
{
    __asm(
        vpdpbusd ymm2, ymm0, ymm1
        vpdpbusd ymm3, ymm0, ymm1
        vpdpbusd ymm4, ymm0, ymm1
        vpdpbusd ymm5, ymm0, ymm1
        vpdpbusd ymm6, ymm0, ymm1
        vpdpbusd ymm7, ymm0, ymm1
        vpdpbusd ymm8, ymm0, ymm1
        vpdpbusd ymm9, ymm0, ymm1
        vpdpbusd ymm10, ymm0, ymm1
        vpdpbusd ymm11, ymm0, ymm1
        vpdpbusd ymm12, ymm0, ymm1
        vpdpbusd ymm13, ymm0, ymm1
    );
}
```

Example 2-4. Class 3 Pseudo-code Snippet

```
while (1)
{
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
        asm(“PAUSE;”)
    );
}
```

2.3.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-threads busy.

2.3.2.3 Scheduling with a Multi-E-Core Module

E-cores within an idle module help improve performance than E-cores in a busy module.

2.3.2.4 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage the scalability and multithread efficiency of E-cores.

2.3.3 RECOMMENDATIONS FOR APPLICATION DEVELOPERS

The following are recommendations when using processors supporting performance hybrid architecture:

- Stay up to date on updates on operating systems and optimized libraries.
- Software must avoid setting hard affinities on either threads or processes to allow the operating system to provide the optimal core selection for Intel Hybrid.
- Software should replace active spin-waits with lightweight waits, ideally using the new UMWAIT/TPAUSE and older PAUSE instructions. This will allow the scheduler to get better hints on time spinning.
- Software can utilize the Windows Power Throttling information using process and thread information APIs to give the scheduler hints on the Quality of Service (QoS) required for a particular thread or process, improving performance and energy efficiency.
- Leverage Windows frameworks and media APIs for multimedia application development. Windows Media Foundation framework is optimized for hybrid architecture, enabling media applications to run efficiently while preventing glitches.
- The Windows IrqPolicyMachineDefault policy enables Windows to target interrupts to the right core optimally, and more so on hybrid architecture.

For additional recommendations and information on performance hybrid architecture, refer to the white papers on the Performance Hybrid Architecture page.

2.4 THE GOLDEN COVE MICROARCHITECTURE

The Golden Cove microarchitecture is the successor of the Ice Lake microarchitecture. The Golden Cove microarchitecture introduces the following enhancements:

- Wider machine: 5→6 wide allocation, 10→12 execution ports, and 4→8 wide retirement.
- Significant increases in the size of key structures enable deeper OOO execution and expose more instruction-level parallelism.
- Greater capabilities per execution port, e.g., 5th integer ALU execution ports with expanded capability and a new fast floating-point adder.
- Intel® Advanced Matrix Extensions (Intel® AMX)[1]: Built-in integrated Tiled Matrix Multiplication / Machine Learning Accelerator.
- Improved branch prediction.
- Improvements for large code footprint workloads, for example, larger branch prediction structures, enhanced code prefetcher, and larger instruction TLB.
- Wider fetch: legacy decode pipeline fetch bandwidth increase to 32B/cycles, 4→6 decoders, increased micro-op cache size, and increased micro-op cache bandwidth.

1. Intel AMX are unavailable on client parts.

- Maximum load bandwidth increased from two loads/cycle to three loads/cycle.
- Larger 4K Pages DTLB, increase of outstanding Page Miss handlers.
- Increase of outstanding misses (16 FB, 32→48 Deeper MLC miss queues).
- Enhanced data prefetchers for increased memory parallelism.
- Mid-level cache size increased to 2MB on server parts; remains 1.25MB on client parts.

2.4.1 GOLDEN COVE MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Golden Cove microarchitecture is depicted in Figure 2-2.

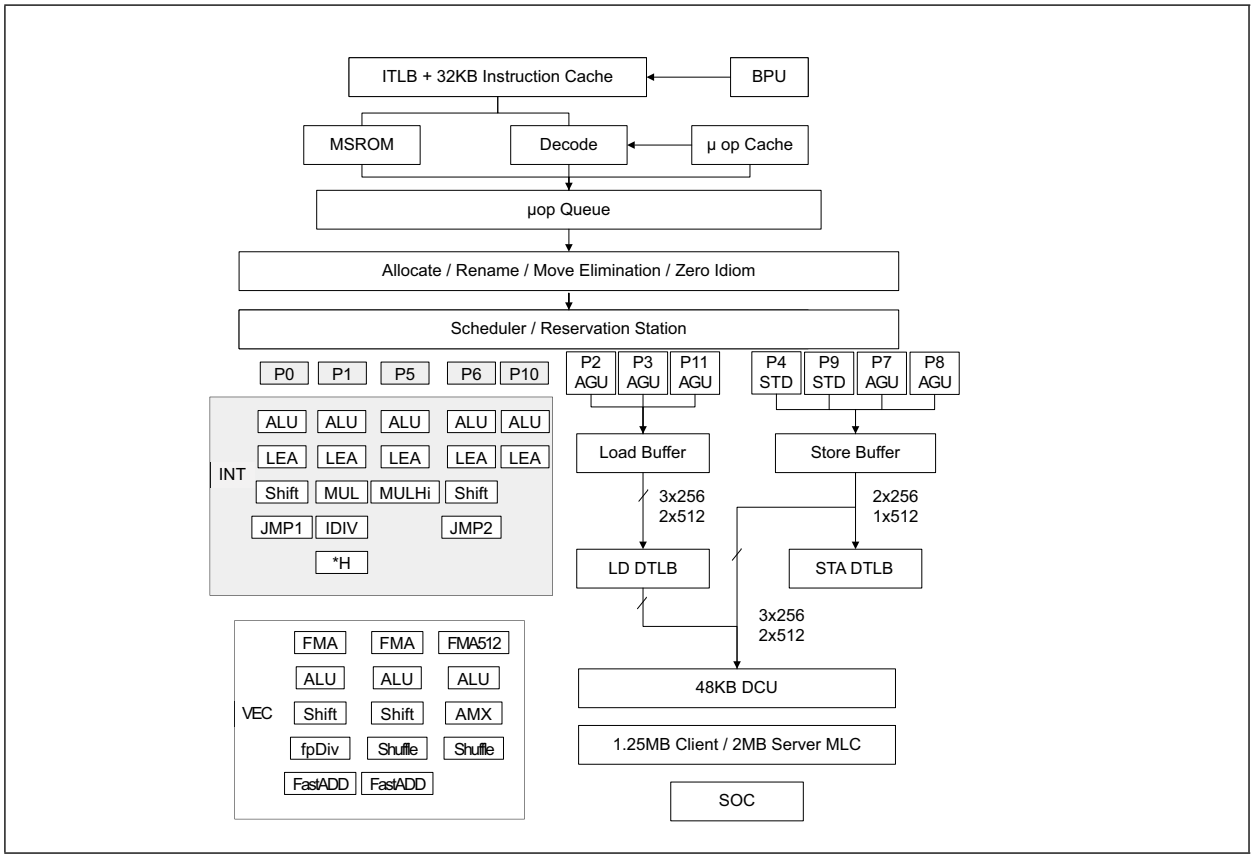


Figure 2-2. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture

The Golden Cove front end is depicted in Figure 2-3. The front end is built to feed the wider and deeper out-of-order core:

- Legacy decode pipeline fetch bandwidth increased from 16 to 32 bytes/cycle.
- The number of decoders increased from four to six, allowing the decode of up to six instructions per cycle.
- The micro-op cache size increased, and its bandwidth increased to deliver up to eight micro-ops per cycle.
- Improved branch prediction.

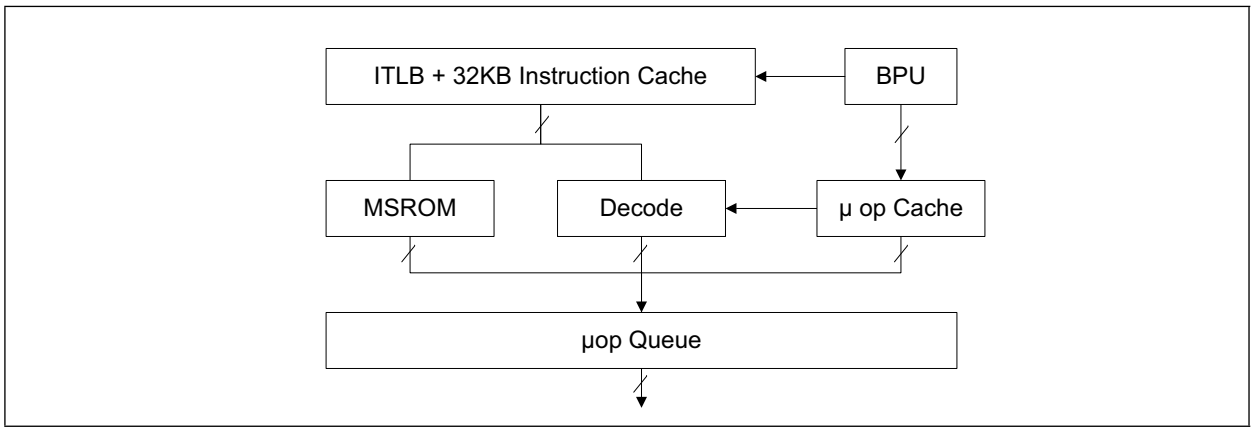


Figure 2-3. Processor Front End of the Golden Cove Microarchitecture

Improvements for large code footprint workloads:

- Double the size of the instruction TLB: 128→256 entries for 4K pages, 16→32 entries for 2M/4M pages.
- Bigger branch prediction structures.
- Enhanced code prefetcher.
- Improved LSD coverage.
- The IDQ can hold 144 µops per logical processor in single-thread mode, or 72 µops per thread when SMT is active.

Additional improvements include:

- The significant increase in the size of key buffer structures to enable deeper OOO execution and expose more instruction-level parallelism.
- Wider machine:
  - Wider allocation (5→6 µops per cycle) and retirement (4→8 µops per cycle) width.
  - Increased execution ports (10→12).
  - Greater capabilities per execution port.

Table 2-3 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU[3] LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture (Contd.)

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. "*" in this table indicates that these features are unavailable for 512-bit vectors.
2. "**" in this table indicates that these features are unavailable for 512-bit vectors in Client parts.
3. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

Table 2-4 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel® SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	5[2]	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2[3]	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	2x256-bit 1x512-bit	(v)add, (v)cmp. (v)max, (v)min, (v)sub, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2sl, (v)cvtss2sl
	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)
Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.
2. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.
3. Ibid.

Table 2-5 describes bypass delay in cycles between producer and consumer operations.

Table 2-5. Bypass Delay Between Producer and Consumer Micro-Ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	SIMD/0,1/1	FMA/0,1/4	MUL/0,1/4	Fast Adder/1,5/3	SIMD/5/1,3	SHUF/ 1,5/1,3	V2I/0/3
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

The attributes relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the µop. For example:

- "SIMD/0,1/1" applies to a 1-cycle vector SIMD µop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle µop dispatched to port 5.
- "V2I/0/3" applies to a three-cycle vector-to-integer µop dispatched to port 0.

- “I2V/5/1” applies to a one-cycle integer-to-vector µop dispatched to port 5.
- “Fast Adder/1,5/3” applies to either a three-cycle 256-bit µop dispatched to port 1 or port 5, or a 512-bit µop dispatched to port 5. This operation supports two cycles back-to-back between a pair of Fast Adder operations.

A new Fast Adder[1] unit is added as 512-bit on port 5 in the VEC stack and as 256-bit on ports 1 and 5. The Fast Adder performs floating-point ADD/SUB operations in three cycles.

Back-to-back ADD/SUB operations, both executed on the Fast Adder unit, perform the operations in two cycles.

- In 128/256-bit, back-to-back ADD/SUB operations executed on the Fast Adder unit perform the operations in two cycles.
- In 512-bit, back-to-back ADD/SUB operations are executed in two cycles if both operations use the Fast Adder unit on port 5.

The following instructions are executed by the Fast Adder unit:

- (V)ADDSUBSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.4.1.1 Cache Subsystem and Memory Subsystem

Changes in the cache subsystem and memory subsystem within the Golden Cove microarchitecture include:

- Maximum load bandwidth increased from two to three loads per cycle. Bandwidth of Intel AVX-512 loads, Intel AMX loads, and MMX/x87 loads remain at a maximum of two loads per cycle.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Number of entries for 4K pages in the load DTLB increased from 64 to 96.
- The Page Miss handler can handle up to four D-side page walks in parallel instead of two.
- Increased outstanding DCU and MLC misses.
- Enhanced data prefetchers for increased memory parallelism.
- Partial store forwarding allowing forwarding data from store to load also when only part of the load was covered by the store (in case the load's offset matches the store's offset).

2.4.1.2 Avoiding Destination False Dependency

Some SIMD instructions incur false dependencies on the destination operand. The following instructions are affected:

- VFMULCSH, VFMULCPH
- VFCMULCSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

1. The Fast Adder unit is unavailable on 512-bit vectors in Client parts.

Recommendation: Use dependency breaking zero idioms on the destination register before the affected instructions to avoid potential slowdown from the false dependency.

Example 2-5. Breaking False Dependency through Zero Idiom

Code with False Dependency Impact	Mitigation: Break False Dependency with Zero Idiom

vaddps zmm3, zmm4, zmm5
vmovaps [rsi], zmm3
vfmulcph zmm3, zmm2, zmm1 ;False dependency on zmm3.
//Will not execute out-of-order until vaddps writes zmm3.
	vaddps zmm3, zmm4, zmm5
	vmovaps [rsi], zmm3
	vpxord zmm3, zmm3, zmm3
	//Dependency-breaking zero idiom.
	vfmulcph zmm3, zmm2, zmm1
	//Execute out-of-order
	without waiting for vaddps result.

2.5 ICE LAKE CLIENT MICROARCHITECTURE

The Ice Lake client microarchitecture introduces the following new features that allow optimizations of applications for performance and power consumption:

- Targeted vector acceleration.
- Crypto acceleration.
- Intel® Software Guard Extensions (Intel® SGX) enhancements.
- Cache line writeback instruction (CLWB).

2.5.1 ICE LAKE CLIENT MICROARCHITECTURE OVERVIEW

The Ice Lake client microarchitecture builds on the successes of the Skylake client microarchitecture. The basic pipeline functionality of the Ice Lake Client microarchitecture is depicted in Figure 2-4.

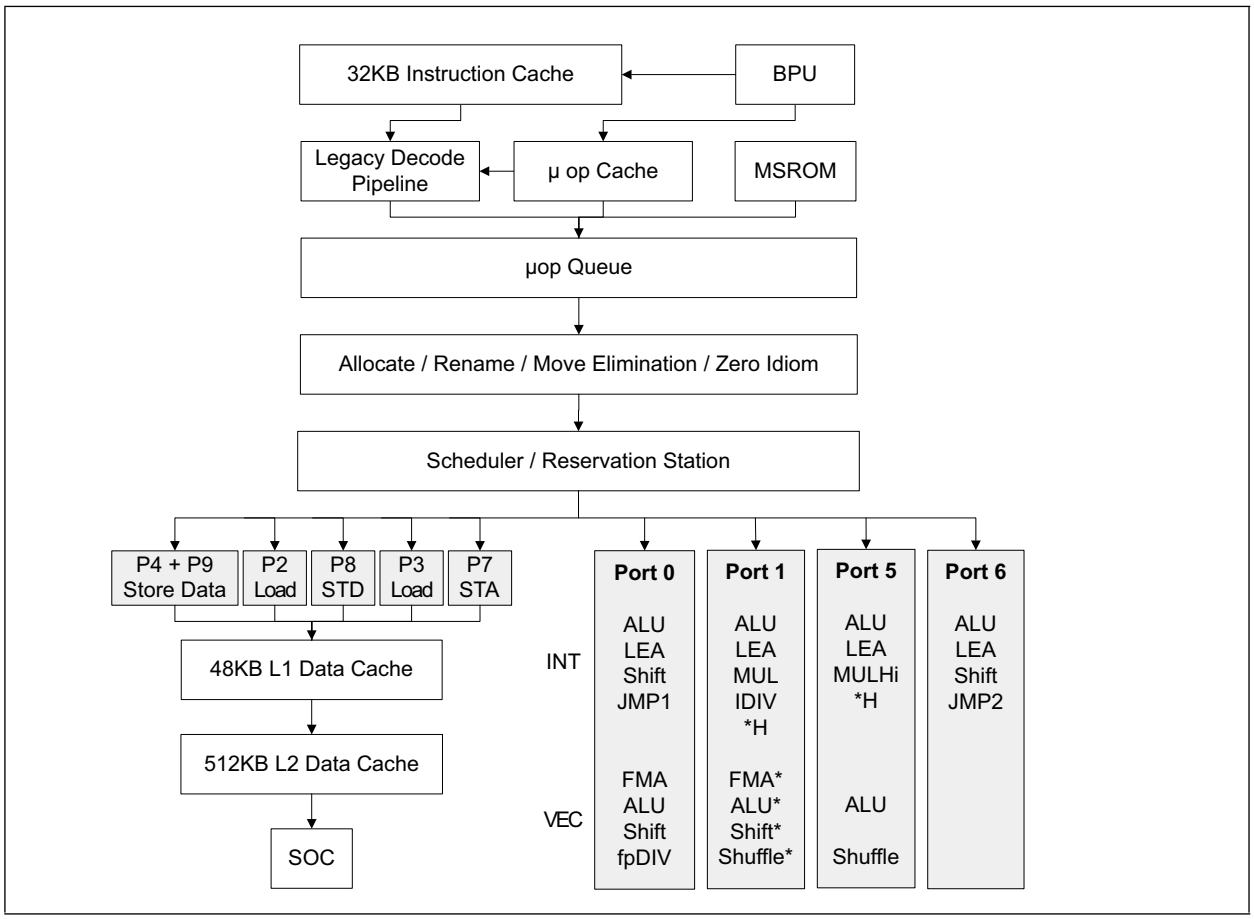


Figure 2-4. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

NOTES:

1. "*" in the figure above indicates these features are unavailable for 512-bit vectors.
2. "INT" represents GPR scalar instructions.
3. "VEC" represents floating-point and integer vector instructions.
4. "MULHi" produces the upper 64 bits of the result of an iMul operation that multiplies two 64-bit registers and places the result into two 64-bits registers.
5. The "Shuffle" on port 1 is new, and supports only in-lane shuffles that operate within the same 128-bit sub-vector.
6. The "IDIV" unit on port 1 is new, and performs integer divide operations at a reduced latency.
7. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

The Ice Lake client microarchitecture introduced the following features:

- Significant increase in size of key structures enable deeper OOO execution.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Intel AVX-512 (new for client processors): 512-bit vector operations, 512-bit loads and stores to memory, and 32 new 512-bit registers.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA), reduced latency Integer Divider.
- 2×BW for AES-NI peak throughput for existing binaries (microarchitectural).

- Rep move string acceleration.
- 50% increase in size of the L1 data cache.
- Reduced effective load latency.
- 2×L1 store bandwidth: 1 → 2 stores per cycle.
- Enhanced data prefetchers for increased memory parallelism.
- Larger 2nd level TLB.
- Larger µop cache.
- Improved branch predictor.
- Large page ITLB size in single thread mode doubled.
- Larger L2 cache.

The Ice Lake client microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of some components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

2.5.1.1 The Front End

The front end changes in Ice Lake Client microarchitecture included:

- Improved branch predictor.
- Large page ITLB in single thread mode increased from 8 to 16 entries.
- Larger µop cache.
- The IDQ can hold 70 µops per logical processor vs. 64 µops per logical processor in previous generations when two sibling logical processors in the same core are active (2×70 vs. 2×64 per core). If only one logical processor is active in the core, the IDQ can hold 70 µops vs. 64 µops.
- The LSD in the IDQ can detect loops of up to 70 µops per logical processor irrespective single thread or multi thread operation.

2.5.1.2 The Out of Order and Execution Engines

The Out of Order and execution engines changes in Ice Lake client microarchitecture include:

- A significant increase in size of reorder buffer, load buffer, store buffer, and reservation stations enable deeper OOO execution and higher cache bandwidth.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA).
- Reduced latency Integer Divider.
- A new iDIV unit was added, significantly reducing the latency and improving the throughput of integer divide operations.

Table 2-6 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-6. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

NOTES:

1. "*" in this table indicates these features are unavailable for 512-bit vectors.

Table 2-7 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	2	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

Table 2-8 describes bypass delay in cycles between producer and consumer operations.

Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the µop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD µop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle µop dispatched to port 5.
- "V2I/0/3" applies to a 3-cycle vector-to-integer µop dispatched to port 0.
- "I2V/5/1" applies to a 1-cycle integer-to-vector µop to port 5.

2.5.1.3 Cache and Memory Subsystem

The cache hierarchy changes in Ice Lake Client microarchitecture include:

- 50% increase in size of the L1 data cache.
- 2×L1 store bandwidth: 3 → 4 AGUs, 1 → 2 store data.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Higher cache bandwidth compared to previous generations.
- Larger 2nd level TLB: 1.5K entries → 2K entries.

- Enhanced data prefetchers for increased memory parallelism.
- L2 cache size increased from 256KB to 512KB.
- L2 cache associativity increased from 4 ways to 8 ways.
- Significant reduction in effective load latency.

Table 2-9. Cache Parameters of the Ice Lake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Latency [1] (cycles)	Peak Bandwidth (bytes/cycles)	Sustained Bandwidth (bytes/cycles)	Update Policy
First Level (DCU)	48KB/8	64	5	2×64B loads + 1x64B or 2x32B stores	Same as peak	Writeback
Second Level (MLC)	512KB/8	64	13	64	48	Writeback
Third Level (LLC)	Up to 2MB per core/up to 16 ways	64	xx[2]	32	21	Writeback

NOTES:

1. Software-visible latency/bandwidth will vary depending on access patterns and other factors.
2. This number depends on core count.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, shared L2 TLB for 4K and 4MB pages and a dedicated L2 TLB for 1GB pages.

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Instruction	4KB	128	64	8
Instruction	2MB/4MB	16	8	8
First Level Data (loads)	4KB	64	64 competitively shared	4
First Level Data (loads)	2MB/4MB	32	32 competitively shared	4
First Level Data (loads)	1GB	8	8 competitively shared	8
First Level Data (stores)	Shared for all page sizes	16	16 competitively shared	16

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture (Contd.)

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Second Level	Shared for all page sizes	2048[1]	2048 competitively shared	16

NOTES:

1. 4K pages can use all 2048 entries. 2/4MB pages can use 1024 entries (in 8 ways), sharing them with 4K pages. 1GB pages can use the other 1024 entries (in 8 ways), also sharing them with 4K pages.

Paired Stores

Ice Lake Client microarchitecture includes two store pipelines in the core, with the following features:

- Two dedicated AGU for LDs on ports 2 and 3.
- Two dedicated AGU for STAs on ports 7 and 8.
- Two fully featured STA pipelines.
- Two 256-bit wide STD pipelines (Intel AVX-512 store data takes two cycles to write).
- Second senior store pipeline to the DCU via store merging.

Ice Lake Client microarchitecture can write two senior stores to the cache in a single cycle if these two stores can be paired together. That is:

- The stores must be to the same cache line.
- Both stores are of the same memory type, WB or USWC.
- None of the stores cross cache line or page boundary.

To maximize performance from the second store port try to:

- Align store operations whenever possible.
- Place consecutive stores in the same cache line (not necessarily as adjacent instructions).

As seen in Example 2-6, it is important to take into consideration all stores, explicit or not.

Example 2-6. Considering Stores

Stores are Paired Across Loop Iterations	Stores Not Paired Due to Stack Update in Between
Loop: compute reg … store [X], reg add X, 4 jmp Loop ; stores from different iterations of the loop can be paired all together because they usually would be same line	Loop: call function to compute reg … store [X], reg add X, 4 jmp Loop ; stores from different iterations of the loop cannot be paired anymore because of the call store to stack ; the call is disturbing pairing

In some cases it is possible to rearrange the code to achieve store pairing. Example 2-7 provides details.

Example 2-7. Rearranging Code to Achieve Store Pairing

Stores to Different Cache Lines - Not Paired	Unrolling May Solve the Problem

Loop:
... compute ymm1 …
vmovaps [x], ymm1
... compute ymm2 …
vmovaps [y], ymm2
add x, 32
add y, 32
jmp Loop
 	
//this loop cannot pair any store because of alternating store to different cache lines [x] and [y]
	Loop:
	... compute ymm1 …
	vmovaps [x], ymm1
	... compute new ymm1 …
	vmovaps [x+32], ymm1
	... compute ymm2 …
	vmovaps [y], ymm2
	... compute new ymm2 …
	vmovaps [y+32], ymm2
	add x, 64
	add y, 64
	jmp Loop
	
	//the loop was unrolled 2 times and stores rearranged to make sure two stores to the same cache line are placed one after another. Now stores to addresses [x] and [x+32] are to the same cache line and could be paired together and executed in same cycle.

2.5.1.4 Fast Store Forwarding Prediction (FSFP)

This section includes recommendations for effective use of Fast Store Forwarding Prediction (FSFP) introduced in Ice Lake microarchitecture. Extrapolated from previous behavior, FSFP enables the processor to predict that a store will forward data to a younger load and optimize that case. The optimization allows the load to complete using the data of predicted store but without accessing the memory. Only integer loads support FSFP in the Ice Lake microarchitecture.

The Fast Store Forwarding Prediction has limitations. to maximize performance gain on Ice Lake microarchitecture it is recommended to follow these recommendations:

- Only loads and stores without Index (encoded with no SIB byte) are supported. LEA operation can be used to avoid Index register usage during memory address computations.
- Loads and stores using RIP-relative addressing do not support FSFP. We recommend using the LEA operation to pre-compute address to enable FSFP for such cases.
- Loads and stores operating with 16-bit General Purpose Registers (AX/BX/CX/DX and etc) or *H 8-bit registers do not support FSFP optimization. We recommend using **movzx** instruction instead of unsupported registers.

Example 2-8. FSFP Optimization

Slow Version Not Enabling PFSP	Enabling FSFP Using LEA Operation

Loop:
mov r10,[rsi+r8*8]
inc qword[rdi+r10*8]
mov r11,[rsi+r8*8]
inc r8
inc qword[rdi+r11*8]
… jmp Loop
	Loop:
	mov r10,[rsi+r8*8]
	lea r12,[rdi+r10*8] ; using LEA to avoid
	;Index register
	for ;inc below
	inc qword[r12]
	mov r11,[rsi+r8*8]
	inc r8
	lea r13,[rdi+r11*8] ; another similar case
	inc qword [r13]
	…. jmp Loop

2.5.1.5 New Instructions

New instructions and architectural changes in Ice Lake Client microarchitecture are listed below. Actual support may be product dependent.

- Crypto acceleration
  - SHA NI for acceleration of SHA1 and SHA256 hash algorithms.
  - Big-Number Arithmetic (IFMA): VPMADD52 - two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance.
  - Galois Field New Instructions (GFNI) for acceleration of various encryption algorithms, error correction algorithms, and bit matrix multiplications.
  - Vector AES and Vector Carry-less Multiply (PCLMULQDQ) instructions to accelerate AES and AES-GCM.
- Security Technologies
  - Intel® SGX enhancements to improve usability and applicability: EDMM, multi-package server support, support for VMM memory oversubscription, performance, larger secure memory.
- Sub Page protection for better performance of security VMMs.
- Targeted Acceleration
  - Vector Bit Manipulation Instructions: VBMI1 (permutes, shifts) and VBMI2 (Expand, Compress, Shifts)- used for columnar database access, dictionary based decompression, discrete mathematics, and data-mining routines (bit permutation and bit-matrix-multiplication).
  - VNNI with support for integer 8 and 16 bits data types- CNN/ML/DL acceleration.
  - Bit Algebra (POPCNT, Bit Shuffle).
  - Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Platform analysis features for more efficient performance software tuning and debug.
  - AnyThread removal.
  - 2x general counters (up to 8 per-thread).
  - Fixed Counter 3 for issue slots.

— New performance metrics for built-in support for Level 1 Top-Down method (% of Issue slots that are front-end bound, back-end bound, bad speculation, retiring) while leaving the 8 general purpose counters free for software use.

2.5.1.6 Ice Lake Client Microarchitecture Power Management

Processors based on Ice Lake client microarchitecture are the first client processors whose cores may execute at a different frequency from one another. The frequency is selected based on the specific instruction mix; the type, width and number of vector instructions of the program that executes on each core, the ratio between active time and idle time of each core, and other considerations such as how many cores share similar characteristics.

Most of the power management features of Skylake Server Microarchitecture (see Section 2.6) is applicable to Ice Lake Client microarchitecture as well. The main differences are the following:

- The typical P0n max frequency difference between Intel® Advanced Vector Extensions (Intel® AVX-512) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) on Ice Lake Client microarchitecture is much lower than on Skylake Server microarchitecture. Therefore, the negative impact on overall application performance is much smaller.
- All processors based on Ice Lake Client microarchitecture contain a single 512-bit FMA unit, whereas some of the processors based on Skylake Server microarchitecture contain two such units. Both processors contain two 256-bit FMA units. The power consumed by Ice Lake Client FMA units is the same, whereas on Skylake Server the 512-bit units consume twice as much.

Compute heavy workloads, especially those that span multiple Ice Lake client cores, execute at a lower frequency than P0n, under Intel AVX-512 and under Intel AVX2 instruction sets, due to power limitations. In this scenario, Intel AVX-512 architecture, which requires less dynamic instructions to complete the same task than Intel AVX2 architecture, consumes less power and thus may achieve higher frequency. The net result may be higher performance due to the shorter path length and a bit higher frequency.

There are still some cases where coding to the Intel AVX-512 instruction set yields lower performance than when coding to the Intel AVX2 instruction set. Sometimes it is due to microarchitecture artifacts of longer vectors, in other cases the natural vectors are just not long enough. Most compilers are still maturing their Intel AVX-512 support, and it may take them a few more years to generate optimal code.

The general recommendation in the Skylake Server Power Management section (see Section 2.6.3) still holds. Developers should code to the Intel AVX-512 instruction set and compare the performance to their Intel AVX2 workload on Ice Lake client microarchitecture, before making the decision to proceed with a complete port.

2.6 SKYLAKE SERVER MICROARCHITECTURE

The Intel® Xeon® Processor scalable processors based on the Skylake microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of CHAPTER 2 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

The Skylake Server microarchitecture introduces the following new features[1] that allow you to optimize your application for performance and power consumption.

- A new core based on the Skylake Server microarchitecture with process improvements based on the Kaby Lake microarchitecture.
- Intel AVX-512 support.
- More cores per socket (max 28 vs. max 22).
- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.

1. Some features may not be available on all products.

- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA).
- Sub-NUMA Clustering (SNC) support.

The gray rectangles in Figure 2-5 represent features different in Skylake Server microarchitecture compared to Skylake microarchitecture for client;

- A 1MB L2 cache.
- An additional Intel AVX-512 FMA unit on port 5 which is available on some parts.

In this figure:

- INTEGER represents GPR scalar instructions.
- VEC represents floating point and integer vector instructions.
- SLOW LEA represents a lea with two registers and displacement, all other lea versions considered as FAST LEA.
- BRANCH1 is the primary branch and more capable than BRANCH2.

Since port 0 and port 1 are 256-bits wide, Intel AVX-512 operations that will be dispatched to port 0 will execute on port 0 and port 1; however, other operations such as **LEA** can still execute on port 1 in parallel. See the red block in Figure 2-9 for the fusion of ports 0 and 1.

Notice that, unlike Skylake microarchitecture for client, the Skylake Server microarchitecture has its front end loop stream detector (LSD) disabled.

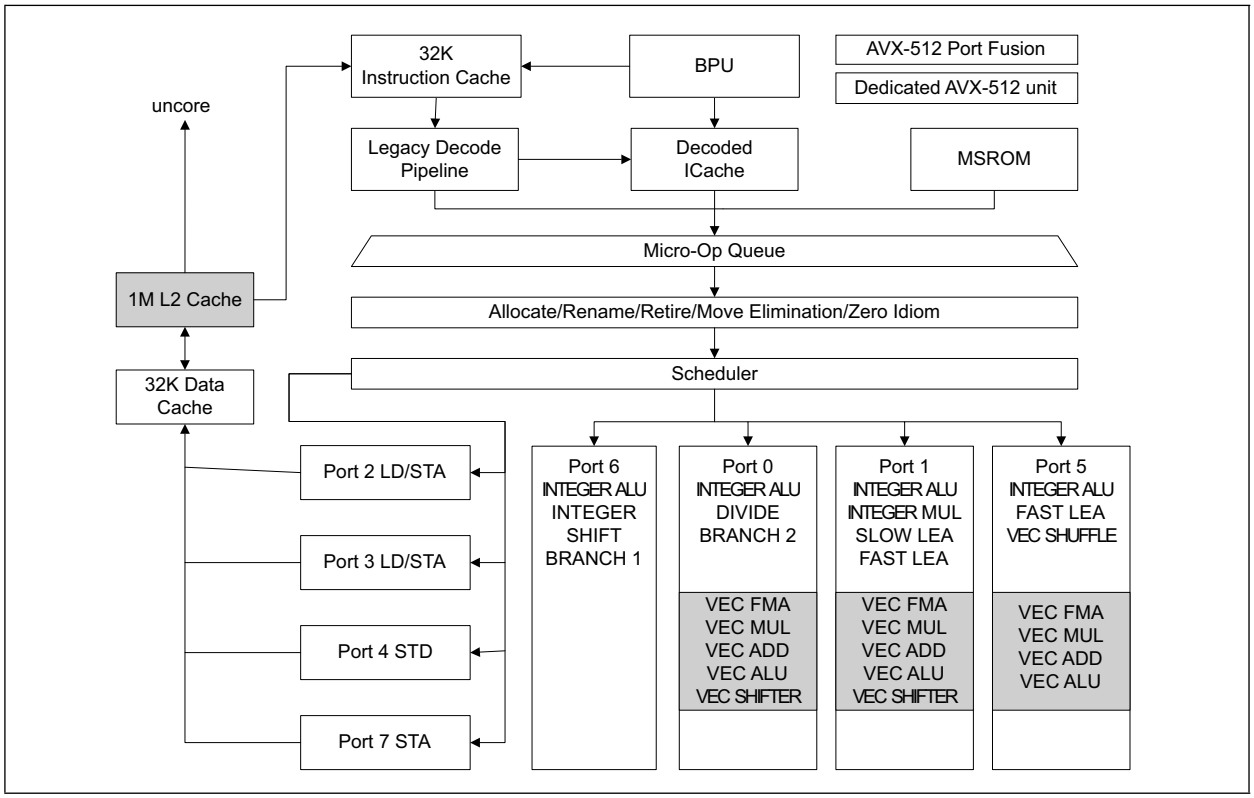


Figure 2-5. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture

2.6.1 SKYLAKE SERVER MICROARCHITECTURE CACHE

Intel Xeon scalable processors based on Skylake server microarchitecture has significant changes in core and uncore architecture to improve performance and scalability of several components compared with the previous generation of the Intel Xeon processors based on the Broadwell microarchitecture.

2.6.1.1 Larger Mid-Level Cache

Skylake server microarchitecture implements a mid-level (L2) cache of 1 MB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64B and it is 16-way associative. The mid-level cache is private to each core.

Software that has been optimized to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake server microarchitecture.

2.6.1.2 Non-Inclusive Last Level Cache

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MBs per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

2.6.1.3 Skylake Server Microarchitecture Cache Recommendations

A high-level comparison between Skylake server microarchitecture cache and the previous generation Broadwell microarchitecture cache is available in the table below.

Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8

Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture (Contd.)

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16
L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375[1] per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	32
	Sustained bandwidth [bytes/cycles]	14	15

NOTES:

1. Some Skylake server parts have some cores disabled and hence have more than 1.375 MBs per core of L3 cache.

The figure below shows how Skylake server microarchitecture shifts the memory balance from shared-distributed with high latency, to private-local with low latency.

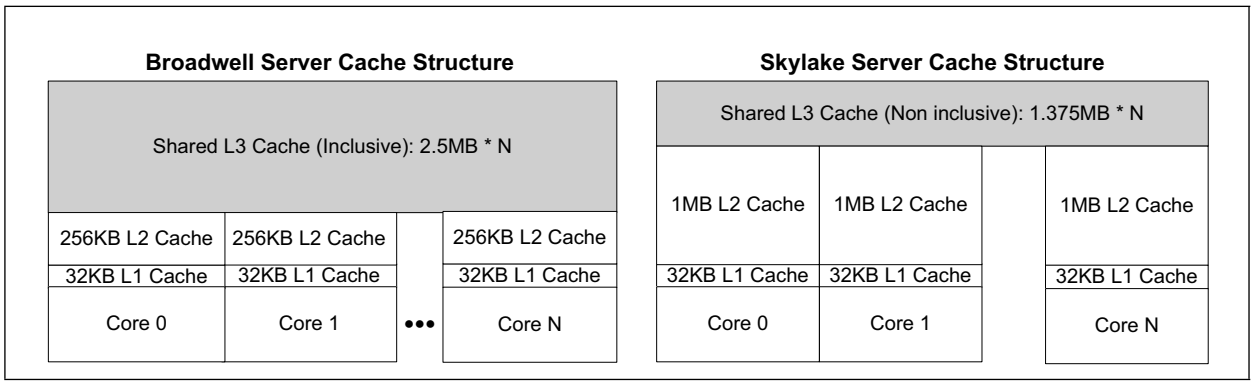


Figure 2-6. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures

The potential performance benefit from the cache changes is high, but software will need to adapt its memory tiling strategy to be optimal for the new cache sizes.

Recommendation: Rebalance application shared and private data sizes to match the smaller, non-inclusive L3 cache, and larger L2 cache.

Choice of cache blocking should be based on application bandwidth requirements and changes from one application to another. Having four times the L2 cache size and twice the L2 cache bandwidth compared to the previous generation Broadwell microarchitecture enables some applications to block to L2 instead of L1 and thereby improves performance.

Recommendation: Consider blocking to L2 on Skylake Server microarchitecture if L2 can sustain the application's bandwidth requirements.

The change from inclusive last level cache to non-inclusive means that the capacity of mid-level and last level cache can now be added together. Programs that determine cache capacity per core at run time should now use a combination of mid-level cache size and last level cache size per core to estimate the effective cache size per core. Using just the last level cache size per core may result in non-optimal use of available on-chip cache; see Section 2.6.2 for details.

Recommendation: In case of no data sharing, applications should consider cache capacity per core as L2 and L3 cache sizes and not only L3 cache size.

2.6.2 NON-TEMPORAL STORES ON SKYLAKE SERVER MICROARCHITECTURE

Because of the change in the size of each bank of last level cache on Skylake server microarchitecture, if an application, library, or driver only considers the last level cache to determine the size of on-chip cache-per-core, it may see a reduction with Skylake server microarchitecture and may use non-temporal store with smaller blocks of memory writes. Since non-temporal stores evict cache lines back to memory, this may result in an increase in subsequent cache misses and memory bandwidth demands on Skylake Server microarchitecture, compared to the previous Intel Xeon processor family.

Also, because of a change in the handling of accesses resulting from non-temporal stores by Skylake Server microarchitecture, the resources within each core remain busy for a longer duration compared to similar accesses on the previous Intel Xeon processor family. As a result, if a series of such instructions are executed, there is a potential that the processor may run out of resources and stall, thus limiting the memory write bandwidth from each core.

The increase in cache misses due to overuse of non-temporal stores and the limit on the memory write bandwidth per core for non-temporal stores may result in reduced performance for some applications.

To avoid the performance condition described above with Skylake server microarchitecture, include mid-level cache capacity per core in addition to the last level cache per core for applications, libraries, or drivers that determine the on-chip cache available with each core. Doing so optimizes the available on-chip cache capacity on Skylake server microarchitecture as intended, with its non-inclusive last level cache implementation.

2.6.3 SKYLAKE SERVER POWER MANAGEMENT

This section describes the interaction of Skylake Server's Power Management and its Vector ISA.

Skylake Server microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics.

Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake Server microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions.

The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution.

Table 2-12 includes information about the maximum Intel® Turbo Boost technology core frequency for each type of instruction executed. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. The more cores belonging to a category at any given time, the lower the maximum frequency.

Table 2-12. Maximum Intel® Turbo Boost Technology Core Frequency Levels

Level	Category	Frequency Level	Max Frequency (P0n)	Instruction Types
0	Intel® AVX2 light instructions	Highest	Max	Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
1	Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions	Medium	Max Intel® AVX2	Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
2	Intel® AVX-512 heavy instructions	Lowest	Max Intel® AVX-512	Intel® AVX-512 FP + INT MUL/FMA

For per SKU max frequency details (reference figure 1-15), refer to the Intel® Xeon® Scalable Processor Family Technical Resources page.

Figure 2-7 is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload.

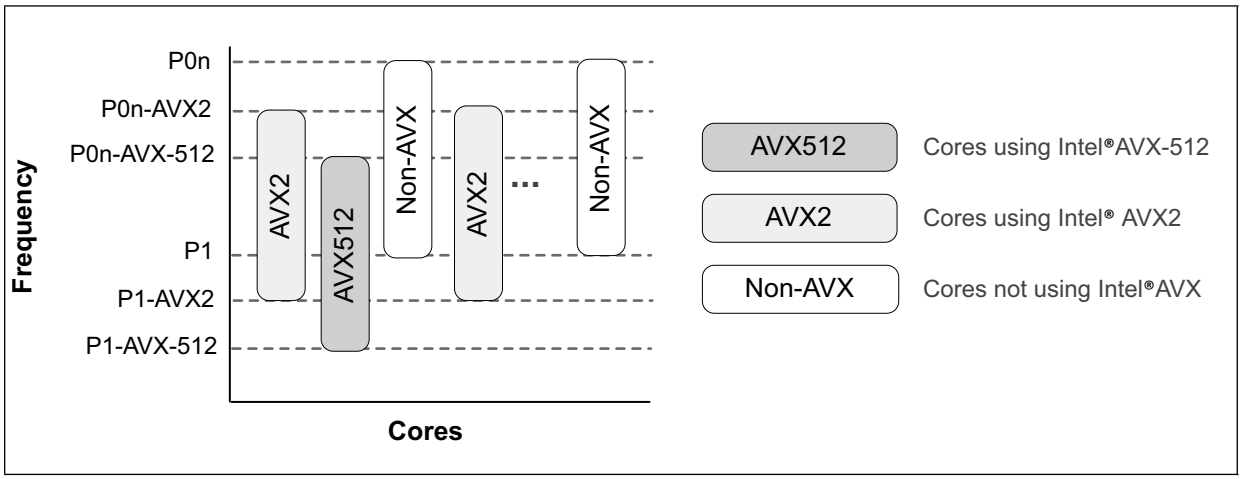


Figure 2-7. Mixed Workloads

The following performance monitoring events can be used to determine how many cycles were spent in each of the three frequency levels.

- CORE_POWER.LVL0_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n.
- CORE_POWER.LVL1_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX2.
- CORE_POWER.LVL2_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX-512.

When the core requests a higher license level than its current one, it takes the PCU up to 500 micro-seconds to grant the new license. Until then the core operates at a lower peak capability. During this time period the PCU evaluates how many cores are executing at the new license level and adjusts their frequency as necessary, potentially lowering the frequency. Cores that execute at other license levels are not affected.

A timer of approximately 2ms is applied before going back to a higher frequency level. Any condition that would have requested a new license resets the timer.

NOTES

A license transition request may occur when executing instructions on a mis-speculated path.

A large enough mix of Intel AVX-512 light instructions and Intel AVX2 heavy instructions drives the core to request License 2, despite the fact that they usually map to License 1. The same is true for Intel AVX2 light instructions and Intel SSE heavy instructions that may drive the core to License 1 rather than License 0. For example, The Intel® Xeon® Platinum 8180 processor moves from license 1 to license 2 when executing a mix of 110 Intel AVX-512 light instructions and 20 256-bit heavy instructions over a window of 65 cycles.

Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the LINPACK benchmark is power limited and does not reach the processor's maximum frequency. The following graph shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

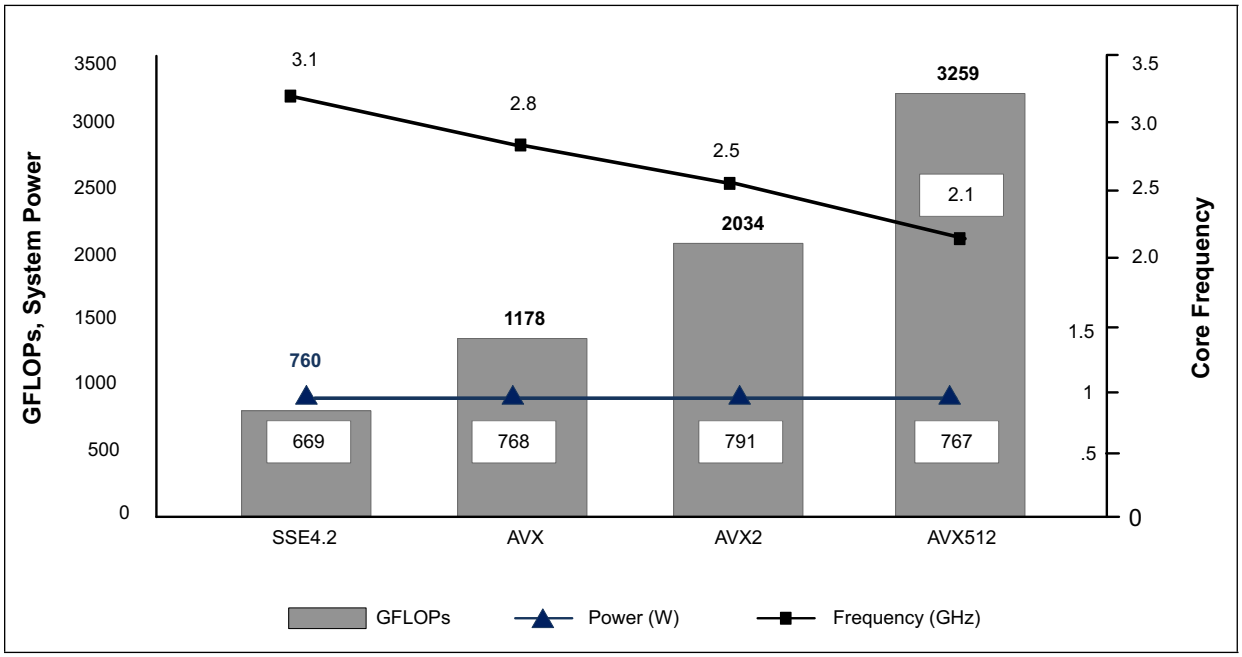


Figure 2-8. LINPACK Performance

Workloads that execute Intel AVX-512 instructions as a large proportion of their whole instruction count can gain performance compared to Intel AVX2 instructions, even though they may operate at a lower frequency. For example, maximum frequency bound Deep Learning workloads that target Intel AVX-512 heavy instructions at a very high percentage can gain 1.3x-1.5x performance improvement vs. the same workload built to target Intel AVX2 (both operating on Skylake Server microarchitecture).

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to the use of zmm registers. However, some programs that use zmm registers may not gain as much, or may even lose performance. It is recommended to try multiple build options and measure the performance of the program.

Recommendation: To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -xCORE-AVX2 **-mtune=skylake-avx512** (Linux* and macOS*)
  /QxCORE-AVX2 **/tune=skylake-avx512** (Windows*)
- -xCORE-AVX512 **-qopt-zmm-usage=low** (Linux* and macOS*)
  /QxCORE-AVX512 **/Qopt-zmm-usage:low** (Windows*)
- -xCORE-AVX512 **-qopt-zmm-usage=high** (Linux* and macOS*)
  /QxCORE-AVX512 **/Qopt-zmm-usage:high** (Windows*)

See Section 17.26 for more information about these options.

The GCC Compiler has the option -mprefer-vector-width=none|128|256|512 to control vector width preference. While -march=skylake-avx512 is designed to provide the best performance for the Skylake Server microarchitecture some programs can benefit from different vector width preferences. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance. -mprefer-vector-width=256 is the default for skylake-avx512.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512
- -march=skylake-avx512 -mprefer-vector-width=512

Clang/LLVM is currently implementing the option -mprefer-vector-width=none|128|256|512, similar to GCC. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512 (plus -mprefer-vector-width=256, if available)
- -march=skylake-avx512 (plus -mprefer-vector-width=512, if available)

2.7 SKYLAKE CLIENT MICROARCHITECTURE

The Skylake client microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake client microarchitecture is depicted in Figure 2-9.

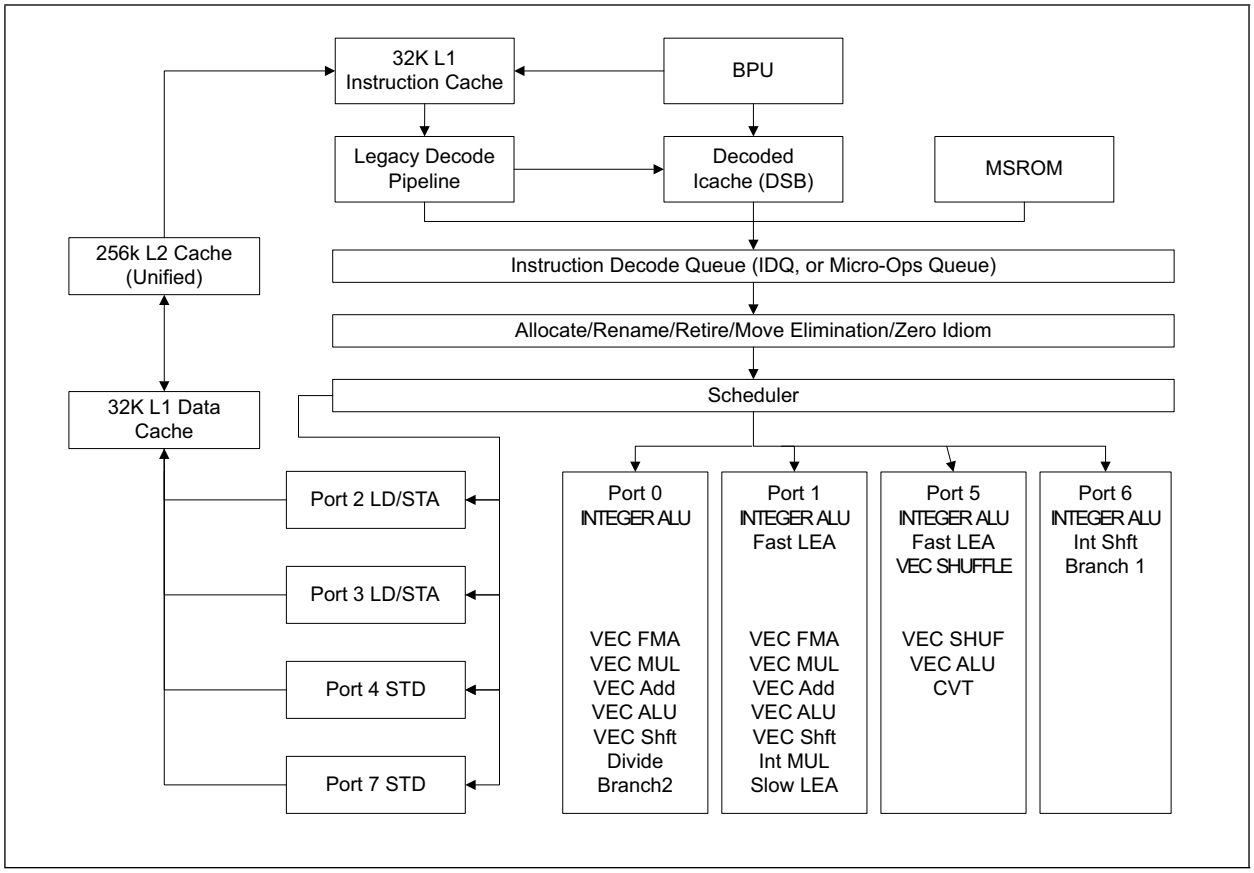


Figure 2-9. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture

The Skylake Client microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 cache (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc.

2.7.1 THE FRONT END

The front end in the Skylake Client microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 µops per cycle to the IDQ compared to 4 µops in previous generations.
- The DSB delivers 6 µops per cycle to the IDQ compared to 4 µops in previous generations.
- The IDQ can hold 64 µops per logical processor vs. 28 µops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 µops (64 vs. 56 µops in ST operation).
- The LSD in the IDQ can detect loops up to 64 µops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.7.2 THE OUT-OF-ORDER EXECUTION ENGINE

The Out of Order and execution engine changes in Skylake Client microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

Table 2-13 summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-13. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

Table 2-14 lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

A significant portion of the Intel SSE, Intel AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two-µop instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

Table 2-15 describes the bypass delay in cycles between a producer µop and the consumer µop. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops (Contd.)

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the µop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD µop dispatched to either port 0 or port 1.
- "VIMUL/0,1/4" applies to 4-cycle vector integer multiply µop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either 1-cycle or 3-cycle non-shuffle µop dispatched to port 5.

2.7.3 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy of the Skylake Client microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-16. Cache Parameters of the Skylake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of Table 2-17 indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-17. TLB Parameters of the Skylake Client Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.7.4 PAUSE LATENCY IN SKYLAKE CLIENT MICROARCHITECTURE

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake Client microarchitecture are listed below.
  - — Front end slots in the Decode ICache, LSD and IDQ.
  - — Execution slots in the RS.
- Save power consumed by the processor core compared with executing equivalent spin loop instruction sequence in the following configurations.
  - — One logical processor is inactive (e.g., entering a C-state).
  - — Both logical processors in the same core execute the PAUSE instruction.
  - — HT is disabled (e.g. using BIOS options).

The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake Client microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

The following is an example of how to use the PAUSE instruction with a dynamic loop iteration count.

Notice that in the Skylake Client microarchitecture the RDTSC instruction counts at the machine's guaranteed P1 frequency independently of the current processor clock (see the INVARIANT TSC property), and therefore, when running in Intel® Turbo-Boost-enabled mode, the delay will remain constant, but the number of instructions that could have been executed will change.

Use Poll Delay function in your lock to wait a given amount of guaranteed P1 frequency cycles, specified in the “clocks” variable.

Example 2-9. Dynamic Pause Loop Example

```
#include <x86intrin.h>
#include <stdint.h>

/* A useful predicate for dealing with timestamps that may wrap.
 Is a before b? Since the timestamps may wrap, this is asking whether it's
 shorter to go clockwise from a to b around the clock-face, or anti-clockwise.
 Times where going clockwise is less distance than going anti-clockwise
 are in the future, others are in the past. e.g. a = MAX-1, b = MAX+1 (=0),
 then a > b (true) does not mean a reached b; whereas signed(a) = -2,
 signed(b) = 0 captures the actual difference */

static inline bool before(uint64_t a, uint64_t b)
{
   return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
   uint64_t endTime = _rdtsc()+ clocks;

   for (; before(_rdtsc(), endTime); )
      _mm_pause();
}
```

For contended spinlocks of the form shown in the baseline example below, we recommend an exponential back off when the lock is found to be busy, as shown in the improved example, to avoid significant performance degradation that can be caused by conflicts between threads in the machine. This is more important as we increase the number of threads in the machine and make changes to the architecture that might aggravate these conflict conditions. In multi-socket Intel server processors with shared memory, conflicts across threads take much longer to resolve as the number of threads contending for the same lock increases. The exponential back off is designed to avoid these conflicts between the threads thus avoiding the potential performance degradation. Note that in the example below,

the number of PAUSE instructions are increased by a factor of 2 until some MAX_BACKOFF is reached which is subject to tuning.

Example 2-10. Contended Locks with Increasing Back-off Example

```
/*******************/
/*Baseline Version */
/*******************/

// atomic {if (lock == free) then change lock state to busy}
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
     __asm__ (“pause”);
   }
}

/*******************/
/*Improved Version */
/*******************/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
  while (lock == busy)
  {
     for (int i=mask; i; --i){
       __asm__ (“pause”);
     }
      mask = mask < max ? mask<<1 : max;
  }
}
```

2.8 INTEL® 64 AND IA-32 INSTRUCTION BEST PRACTICES

2.8.1 NON-PRIVILEGED INSTRUCTION SERIALIZATION

Software may be required to serialize the execution pipeline of the current processor for various reasons in parallel execution environments. Several defined instructions, privileged and non-privileged, are classified as serializing instructions.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution. It is important to note that executing serializing instructions on P6 and more recent processor families constrains speculative execution because the results of speculatively executed instructions are discarded.

These instructions ensure the processor completes all modifications to flags, registers, and memory by previous instructions and drains all buffered writes to memory before the next instruction is fetched and executed. The non-privileged serialization instructions are:

- SERIALIZE
- CPUID
- IRET
- RSM

The SERIALIZE instruction was introduced in the Sapphire Rapids and Alder Lake platforms as a purpose-specific method of providing serialization to supersede the current typical usages such as CPUID.(EAX=0H). For example, CPUID usage for serialization has issues such that registers [EAX, EBX, ECX, EDX] are modified and, when executed on top of a VMM, will always incur the latency of a VM exit/VM entry round trip. SERIALIZE does not modify registers, arithmetic flags, or memory and does not incur a VM exit. The **SERIALIZE i**nstruction is enumerated via **CPUID.(EAX=07H,ECX=0):EDX[14]=1**, and software must verify support before usage.

Software that uses CPUID for serialization is recommended to use **Leaf 0 [CPUID.(EAX=0H)] CPUID**. CPUID leaves have variable performance with Leaf 0 providing the lowest latency when executed natively.

2.8.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT TECHNOLOGY)

Intel® Hyper-Threading Technology (Intel® HT Technology) enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package, or within each processor core in a physical processor package. In its first implementation in the Intel® Xeon® processor, Intel HT Technology makes a single physical processor (or a processor core) appear as two or more logical processors.

Most Intel Architecture processor families support Intel HT Technology with two logical processors in each processor core, or in a physical processor in early implementations. The rest of this section describes features of the early implementation of Intel HT Technology. Most of the descriptions also apply to later implementations supporting two logical processors. The microarchitecture sections in this chapter provide additional details to individual microarchitecture and enhancements to Intel HT Technology.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an Intel HT Technology-capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, Intel HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

Figure 2-10 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting Intel HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an “add” operation from logical processor 0 and another “add” operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of Intel HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing Intel HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

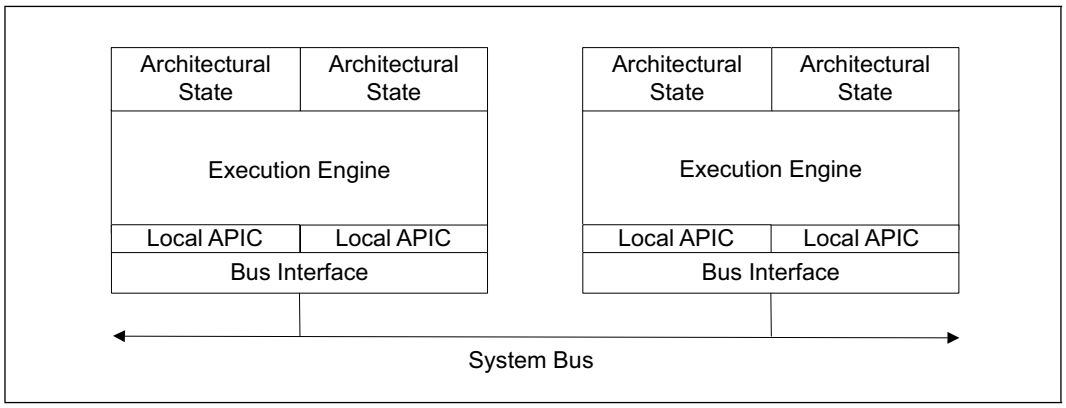


Figure 2-10. Intel® Hyper-Threading Technology on an SMP System

The performance potential due to Intel HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.8.2.1 Processor Resources and Intel® HT Technology

Most microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D.

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the two-entry instruction streaming buffers) were replicated to reduce complexity.

Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include µop queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

2.8.2.2 Microarchitecture Pipeline and Intel® HT Technology

This section describes the Intel HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage. Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.8.2.3 Execution Core

The core can dispatch up to six µops per cycle, provided the µops are ready to execute. Once the µops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned so each uses half the entries.

2.8.2.4 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor must write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.8.3 SIMD TECHNOLOGY

SIMD computations (see Figure 2-11) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-12).

Earlier processors extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-12). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-11 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

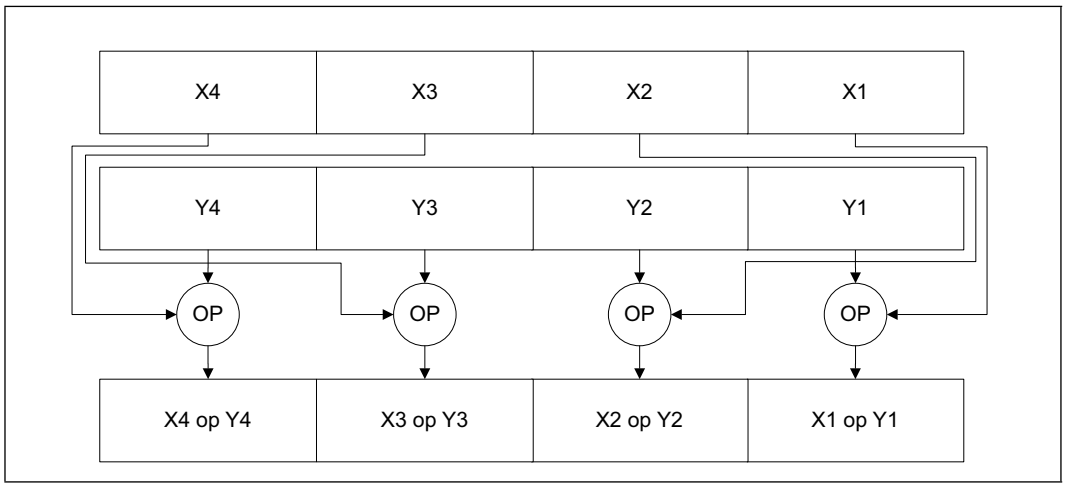


Figure 2-11. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-12.

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

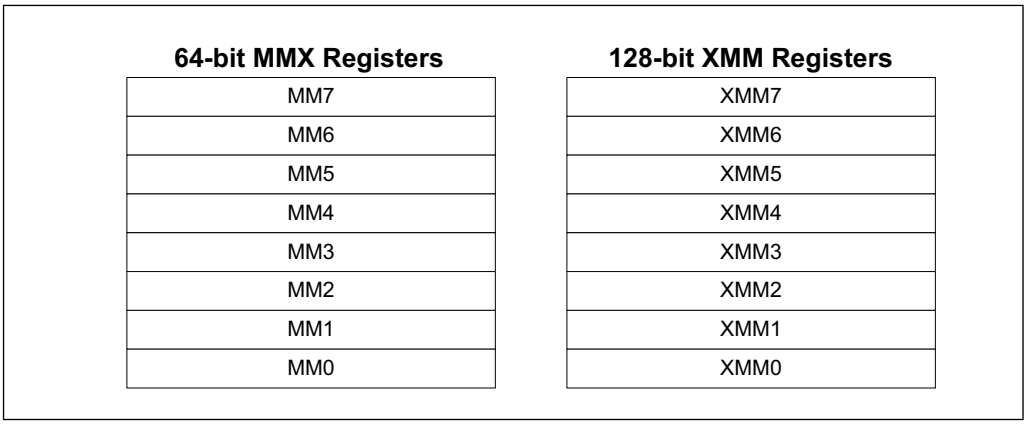


Figure 2-12. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.8.4 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1

- Chapter 9, "Programming with Intel® MMX™ Technology."
- Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."
- Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."
- Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI."
- Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2."
- Chapter 15, "Programming with Intel® AVX-512."

2.8.4.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

Recommendation: Integer SIMD code written using MMX instructions should consider more efficient implementations using SSE/Intel AVX instructions.

2.8.4.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.8.4.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.
- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.8.4.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).
- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.8.4.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.

- 6 instructions that evaluate the absolute values.
- 2 instructions that perform multiply and add operations and speed up the evaluation of dot products.
- 2 instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- 2 instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- 6 instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- 2 instructions that align data from the composite of two operands.

2.8.5 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction provides a streaming hint for WC loads.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations of word integers.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

2.8.5.1 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
  - Basic and advanced string library functions from strlen, strcmp, to strcspn.
  - Delimiter processing, token extraction for lexing of text streams.
  - Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.8.5.2 AESNI and PCLMULQDQ

AESNI introduces seven new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES iteration.

AES encryption involves processing 128-bit input data (plain text) through a finite number of iterative operation, referred to as “AES round”, into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the “equivalent inverse cipher” instead of the “inverse cipher”.

The cryptographic processing at each round involves two input data, one is the “state”, the other is the “round key”. Each round uses a different “round key”. The round keys are derived from the cipher key using a “key schedule” algorithm. The “key schedule” algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.8.5.3 Intel® Advanced Vector Extensions (Intel® AVX)

Intel® Advanced Vector Extensions (Intel® AVX) offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

Intel AVX instruction set and 256-bit register state management detail are described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D. Optimization techniques for Intel AVX are discussed in Chapter 15, “Programming with Intel® AVX-512.”

2.8.5.4 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.8.5.5 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.8.5.6 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and

multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.8.5.7 Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, Intel AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.8.5.8 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. Most of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

There instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 and chapters 3, 4 and 5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

2.8.5.9 RDSEED

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.8.5.10 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics.

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

This chapter discusses general optimization techniques that can improve the performance of applications running on Intel® processors. These techniques take advantage of microarchitectural features described in Chapter 2, "Intel® 64 and IA-32 Processor Architectures." Optimization guidelines focusing on Intel multi-core processors, Hyper-Threading Technology, and 64-bit mode applications are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (Intel® HT)," and Chapter 13, "64-bit Mode Coding Guidelines."

Practices that optimize performance focus on three areas:

- Tools and techniques for code generation.
- Analysis of the performance characteristics of the workload and its interaction with microarchitectural sub-systems.
- Tuning code to the target microarchitecture (or families of microarchitecture) to improve performance.

Some hints on using tools are summarized first to simplify the first two tasks. The rest of the chapter will focus on recommendations for code generation or code tuning to the target microarchitectures.

This chapter explains optimization techniques for the Intel® C++ Compiler, the Intel® Fortran Compiler, and other compilers.

3.1 PERFORMANCE TOOLS

Intel offers several tools to help optimize application performance, including compilers, performance analysis, and multithreading tools.

3.1.1 INTEL® C++ AND FORTRAN COMPILERS

Intel compilers support multiple operating systems (Windows*, Linux*, Mac OS*, and embedded). The Intel compilers optimize performance and give application developers access to advanced features, including:

- Flexibility to target 32-bit or 64-bit Intel processors for optimization.
- Compatibility with many integrated development environments or third-party compilers.
- Automatic optimization features to take advantage of the target processor's architecture.
- Automatic compiler optimization reduces the need to write different code for different processors.
- Common compiler features that are supported across Windows, Linux, and Mac OS include:
  - General optimization settings.
  - Cache-management features.
  - Interprocedural optimization (IPO) methods.
  - Profile-guided optimization (PGO) methods.
  - Multithreading support.
  - Floating-point arithmetic precision and consistency support.
  - Compiler optimization and vectorization reports.

3.1.2 GENERAL COMPILER RECOMMENDATIONS

Generally speaking, a compiler tuned for a target microarchitecture can be expected to match or outperform hand-coding. However, if performance problems are noted with the compiled code, some compilers (like Intel C++ and Fortran compilers) allow the coder to insert intrinsics or inline assembly to exert control over generated code. If inline assembly is used, the user must verify that the code generated is high quality and yields good performance.

Default compiler switches are targeted for common cases. An optimization may be made to the compiler default if it benefits most programs. If the root cause of a performance problem is a poor choice on the part of the compiler, using different switches or compiling the targeted module with a different compiler may be the solution. See the "Quick Reference Guide to Optimization with Intel® C++ and Fortran Compilers" for additional suggestions on compiler Optimization Options, including processor-specific ones.

3.1.3 VTUNE™ PERFORMANCE ANALYZER

VTune uses performance monitoring hardware to collect statistics and coding information about your application and its interaction with the microarchitecture. This allows software engineers to measure performance characteristics of the workload for a given microarchitecture. VTune supports all current and past Intel processor families.

The VTune Performance Analyzer provides two kinds of feedback:

- Indication of a performance improvement gained by using a specific coding recommendation or microarchitectural feature.
- Information on whether a change in the program has improved or degraded performance with respect to a particular metric.

The VTune Performance Analyzer also provides measures for a number of workload characteristics, including:

- Retirement throughput of instruction execution as an indication of the degree of extractable instruction-level parallelism in the workload.
- Data traffic locality as an indication of the stress point of the cache and memory hierarchy.
- Data traffic parallelism as an indication of the degree of effectiveness of amortization of data access latency.

NOTE

Improving performance in one part of the machine does not necessarily bring significant gains to overall performance. It is possible to degrade overall performance by improving performance for some particular metric.

Where appropriate, coding recommendations in this chapter include descriptions of the VTune Performance Analyzer events that provide measurable data on the performance gain achieved by following the recommendations. For more on using the VTune analyzer, refer to the application's online help.

3.2 PROCESSOR PERSPECTIVES

Many coding recommendations work well across current microarchitectures. However, there are situations where a recommendation may benefit one microarchitecture more than another.

3.2.1 CPUID DISPATCH STRATEGY AND COMPATIBLE CODE STRATEGY

When optimum performance on all processor generations is desired, applications can take advantage of the **CPUID** instruction to identify the processor generation and integrate processor-specific instructions into the source code.

The Intel C++ Compiler supports the integration of different versions of the code for different target processors. The selection of which code to execute at runtime is made based on the CPU identifiers. Binary code targeted for different processor generations can be generated under the control of the programmer or by the compiler. Refer to the "Intel® C++ Compiler Classic Developer Guide and Reference" cpu_dispatch and cpu_specific sections for more information on CPU dispatching (a.k.a function multi-versioning).

For applications that target multiple generations of microarchitectures, and where minimum binary code size and single code path is important, a compatible code strategy is the best. Optimizing applications using techniques developed for the Intel Core microarchitecture combined with Nehalem microarchitecture are likely to improve code efficiency and scalability when running on processors based on current and future generations of Intel 64 and IA-32 processors.

3.2.2 TRANSPARENT CACHE-PARAMETER STRATEGY

If the **CPUID** instruction supports function leaf 4, also known as deterministic cache parameter leaf, the leaf reports cache parameters for each level of the cache hierarchy in a deterministic and forward-compatible manner across Intel 64 and IA-32 processor families.

For coding techniques that rely on specific parameters of a cache level, using the deterministic cache parameter allows software to implement techniques in a way that is forward-compatible with future generations of Intel 64 and IA-32 processors, and cross-compatible with processors equipped with different cache sizes.

3.2.3 THREADING STRATEGY AND HARDWARE MULTITHREADING SUPPORT

Intel 64 and IA-32 processor families offer hardware multithreading support in two forms: multi-core technology and HT Technology.

To fully harness the performance potential of hardware multithreading in current and future generations of Intel 64 and IA-32 processors, software must embrace a threaded approach in application design. At the same time, to address the widest range of installed machines, multithreaded software should be able to run without failure on a single processor without hardware multithreading support and should achieve performance on a single logical processor that is comparable to an unthreaded implementation (if such comparison can be made). This generally requires architecting a multithreaded application to minimize the overhead of thread synchronization. Additional guidelines on multithreading are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (Intel® HT)."

3.3 CODING RULES, SUGGESTIONS, AND TUNING HINTS

This section includes rules, suggestions, and hints. They are targeted for engineers who are:

- Modifying source code to enhance performance (user/source rules).
- Writing assemblers or compilers (assembly/compiler rules).
- Doing detailed performance tuning (tuning suggestions).

Coding recommendations are ranked in importance using two measures:

- Local impact (high, medium, or low) refers to a recommendation's affect on the performance of a given instance of code.
- Generality (high, medium, or low) measures how often such instances occur across all application domains. Generality may also be thought of as "frequency."

These recommendations are approximate. They can vary depending on coding style, application domain, and other factors.

The purpose of the high, medium, and low (H, M, and L) priorities is to suggest the relative level of performance gain one can expect if a recommendation is implemented.

Because it is not possible to predict the frequency of a particular code instance in applications, priority hints cannot be directly correlated to application-level performance gain. In cases in which application-level performance gain has been observed, we have provided a quantitative characterization of the gain (for information only). In cases in which the impact has been deemed inapplicable, no priority is assigned.

3.4 OPTIMIZING THE FRONT END

Optimizing the front end covers two aspects:

- Maintaining steady supply of micro-ops to the execution engine — Mispredicted branches can disrupt streams of micro-ops, or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit. Common techniques are covered in Section 3.4.1
- Supplying streams of micro-ops to utilize the execution bandwidth and retirement bandwidth as much as possible. In Sandy Bridge microarchitecture, this aspect focuses on keeping the hot code running from Decoded ICache. Techniques to maximize decode throughput for Intel microarchitecture are covered in Section 3.4.2

3.4.1 BRANCH PREDICTION OPTIMIZATION

Branch optimizations have a significant impact on performance. By understanding the flow of branches and improving their predictability, you can increase the speed of code significantly.

Optimizations that help branch prediction are:

- It is critical to keep code and data on separate pages. See Section 3.6 for more information.
- Eliminate branches whenever possible.
- Arrange code to be consistent with the static branch prediction algorithm.
- Use the **PAUSE** instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase).
- Avoid putting multiple conditional branches in the same 8-byte aligned code block (i.e, have their last bytes' addresses within the same 8-byte aligned code) if the lower 6 bits of their target IPs are the same. This restriction has been removed in Ice Lake Client and later microarchitectures.

3.4.1.1 Eliminating Branches

Eliminating branches improves performance because:

- It reduces the possibility of mispredictions.
- It reduces the number of required branch target buffer (BTB) entries. Conditional branches that are never taken do not consume BTB resources.

There are four principal ways of eliminating branches:

- Arrange code to make basic blocks contiguous.

- Unroll loops, as discussed in Section 3.4.1.6
- Use the **CMOV** instruction.
- Use the **SETCC** instruction.

The following rules apply to branch elimination:

Assembly/Compiler Coding Rule 1. (MH impact, M generality) *Arrange code to make basic blocks contiguous and eliminate unnecessary branches.*

Assembly/Compiler Coding Rule 2. (M impact, ML generality) *Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.*

Consider a line of C code that has a condition dependent upon one of the constants:

X = (A < B) ? CONST1 : CONST2;

This code conditionally compares two values, **A** and **B**. If the condition is true, **X** is set to **CONST1**; otherwise it is set to **CONST2**. An assembly code sequence equivalent to the above C code can contain branches that are not predictable if there are no correlation in the two values.

Example 3-1 shows the assembly code with unpredictable branches. The unpredictable branches can be removed with the use of the **SETCC** instruction. Example 3-2 shows optimized code that has no branches.

Example 3-1. Assembly Code With an Unpredictable Branch

```
    cmp a, b                ; Condition
    jbe L30                 ; Conditional branch
    mov ebx const1          ; ebx holds X
    jmp L31                 ; Unconditional branch
L30:
    mov ebx, const2
L31:
```

Example 3-2. Code Optimization to Eliminate Branches

```
xor   ebx, ebx      ; Clear ebx (X in the C code)
cmp   A, B
setge bl            ; When ebx = 0 or 1
                    ; OR the complement condition
sub   ebx, 1        ; ebx=11...11 or 00...00
and   ebx, CONST3; CONST3 = CONST1-CONST2
add   ebx, CONST2; ebx=CONST1 or CONST2
```

The optimized code in Example 3-2 sets **EBX** to zero, then compares A and B. If A is greater than or equal to B, **EBX** is set to one. Then **EBX** is decreased and **AND**'d with the difference of the constant values. This sets **EBX** to either zero or the difference of the values. By adding **CONST2** back to **EBX**, the correct value is written to **EBX**. When **CONST2** is equal to zero, the last instruction can be deleted.

Another way to remove branches is to use the **CMOV** and **FCMOV** instructions. Example 3-3 shows how to change a **TEST** and branch instruction sequence using **CMOV** to eliminate a branch. If the **TEST** sets the equal flag, the value in **EBX** will be moved to **EAX**. This branch is data-dependent, and is representative of an unpredictable branch.

Example 3-3. Eliminating Branch with CMOV Instruction

```
    test ecx, ecx
    jne  1H
    mov  eax, ebx

1H:
; To optimize code, combine jne and mov into one cmovcc instruction that checks the equal flag
    test     ecx, ecx              ; Test the flags
    cmoveq   eax, ebx              ; If the equal flag is set, move
                                   ; ebx to eax- the 1H: tag no longer needed
```

An extension to this concept can be seen in the AVX-512 masked operations, as well as in some instructions such as VPCMP which can be used to eliminate data dependent branches; see Section 17.4.

3.4.1.2 Static Prediction

Branches that do not have a history in the BTB (see Section 3.4.1) are predicted using a static prediction algorithm:

- Predict forward conditional branches to be NOT taken.
- Predict backward conditional branches to be taken.
- Predict indirect branches to be NOT taken.

The following rule applies to static prediction:

Assembly/Compiler Coding Rule 3. (M impact, H generality) *Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.*

Example 3-4 illustrates the static branch prediction algorithm. The body of an **IF-THEN** conditional is predicted.

Example 3-4. Static Branch Prediction Algorithm

```
//Forward condition branches not taken (fall through)
    IF<condition> {....
    ↓
    }

IF<condition> {...
    ↓
    }

//Backward conditional branches are taken
    LOOP {...
    ↑ –– }<condition>

//Unconditional branches taken
     JMP
    ------→
```

Example 3-5 and Example 3-6 provide basic rules for a static prediction algorithm. In Example 3-5, the backward branch (**JC BEGIN**) is not in the BTB the first time through; therefore, the BTB does not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

Example 3-5. Static Taken Prediction

```
Begin:  mov     eax,  mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
```

The first branch instruction (**JC BEGIN**) in Example 3-6 is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through. The static prediction algorithm correctly predicts that the **CALL CONVERT** instruction will be taken, even before the branch has any branch history in the BTB.

Example 3-6. Static Not-Taken Prediction

```
        mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
        mov     eax, 0
Begin:  call    Convert
```

The Intel Core microarchitecture does not use the static prediction heuristic. However, to maintain consistency across Intel 64 and IA-32 processors, software should maintain the static prediction heuristic as the default.

3.4.1.3 Inlining, Calls, and Returns

The return address stack mechanism augments the static and dynamic predictors to optimize specifically for calls and returns. It holds 16 entries, which is large enough to cover the call depth of most programs. If there is a chain of more than 16 nested calls and more than 16 returns in rapid succession,
performance may degrade.

To enable the use of the return stack mechanism, calls and returns must be matched in pairs. If this is done, the likelihood of exceeding the stack depth in a manner that will impact performance is very low.

The following rules apply to inlining, calls, and returns:

Assembly/Compiler Coding Rule 4. (MH impact, MH generality) *Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns.*

Calls and returns are expensive; use inlining for the following reasons:

- Parameter passing overhead can be eliminated.
- In a compiler, inlining a function exposes more opportunity for optimization.
- If the inlined routine contains branches, the additional context of the caller may improve branch prediction within the routine.
- A mispredicted branch can lead to performance penalties inside a small function that are larger than those that would occur if that function is inlined.

Assembly/Compiler Coding Rule 5. (MH impact, MH generality) *Selectively inline a function if doing so decreases code size or if the function is small and the call site is frequently executed.*

Assembly/Compiler Coding Rule 6. (ML impact, ML generality) *If there are more than 16 nested calls and returns in rapid succession; consider transforming the program with inline to reduce the call depth.*

Assembly/Compiler Coding Rule 7. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred.*

Assembly/Compiler Coding Rule 8. (L impact, L generality) *If the last statement in a function is a call to another function, consider converting the call to a jump. This will save the call/return overhead as well as an entry in the return stack buffer.*

Assembly/Compiler Coding Rule 9. (M impact, L generality) *Do not put more than four branches in a 16-byte chunk.*

Assembly/Compiler Coding Rule 10. (M impact, L generality) *Do not put more than two end loop branches in a 16-byte chunk.*

3.4.1.4 Code Alignment

Careful arrangement of code can enhance cache and memory locality. Likely sequences of basic blocks should be laid out contiguously in memory. This may involve removing unlikely code, such as code to handle error conditions, from the sequence. See Section 3.7 on optimizing the instruction prefetcher.

Assembly/Compiler Coding Rule 11. (M impact, H generality) *When executing code from the Decoded ICache, direct branches that are mostly taken should have all their instruction bytes in a 64B cache line and nearer the end of that cache line. Their targets should be at or near the beginning of a 64B cache line.*

When executing code from the legacy decode pipeline, direct branches that are mostly taken should have all their instruction bytes in a 16B aligned chunk of memory and nearer the end of that 16B aligned chunk. Their targets should be at or near the beginning of a 16B aligned chunk of memory.

Assembly/Compiler Coding Rule 12. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, it should be placed on a different code page.*

3.4.1.5 Branch Type Selection

The default predicted target for indirect branches and calls is the fall-through path. Fall-through prediction is overridden if and when a hardware prediction is available for that branch. The predicted branch target from branch prediction hardware for an indirect branch is the previously executed branch target.

The default prediction to the fall-through path is only a significant issue if no branch prediction is available, due to poor code locality or pathological branch conflict problems. For indirect calls, predicting the fall-through path is usually not an issue, since execution will likely return to the instruction after the associated return.

Placing data immediately following an indirect branch can cause a performance problem. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations and this can cause resource conflicts and slow down branch recovery. Also, data immediately following indirect branches may appear as branches to the branch predication hardware, which can branch off to execute other data pages. This can lead to subsequent self-modifying code problems.

Assembly/Compiler Coding Rule 13. (M impact, L generality) *When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path.*

Indirect branches resulting from code constructs (such as switch statements, computed **GOTO**s or calls through pointers) can jump to an arbitrary number of locations. If the code sequence is so the target destination of a branch

goes to the same address most of the time, then the BTB will predict accurately most of the time. Since only one taken (non-fall-through) target can be stored in the BTB, indirect branches with multiple taken targets may have lower prediction rates.

The effective number of targets stored may be increased by introducing additional conditional branches. Adding a conditional branch to a target is fruitful if:

- The branch direction is correlated with the branch history leading up to that branch; that is, not just the last target, but how it got to this branch.
- The source/target pair is common enough to warrant using the extra branch prediction capacity. This may increase the number of overall branch mispredictions, while improving the misprediction of indirect branches. The profitability is lower if the number of mispredicting branches is very large.

User/Source Coding Rule 1. (M impact, L generality) *If an indirect branch has two or more common taken targets and at least one of those targets is correlated with branch history leading up to the branch, then convert the indirect branch to a tree where one or more indirect branches are preceded by conditional branches to those targets. Apply this "peeling" procedure to the common target of an indirect branch that correlates to branch history.*

The purpose of this rule is to reduce the total number of mispredictions by enhancing the predictability of branches (even at the expense of adding more branches). The added branches must be predictable for this to be worthwhile. One reason for such predictability is a strong correlation with preceding branch history. That is, the directions taken on preceding branches are a good indicator of the direction of the branch under consideration.

Example 3-7 shows a simple example of the correlation between a target of a preceding conditional branch and a target of an indirect branch.

Example 3-7. Indirect Branch with Two Favored Targets

```
function ()
{
int n = rand();          // random integer 0 to RAND_MAX
    if ( ! (n & 0x01) ) { // n will be 0 half the times
          n = 0;          // updates branch history to predict taken
    }
    // indirect branches with multiple taken targets
    // may have lower prediction rates

 switch (n) {
    case 0: handle_0(); break; // common target, correlated with
                               // branch history that is forward taken
    case 1: handle_1(); break; // uncommon
    case 3: handle_3(); break; // uncommon
    default: handle_other();   // common target
     }
}
```

Correlation can be difficult to determine analytically, for a compiler and for an assembly language programmer. It may be fruitful to evaluate performance with and without peeling to get the best performance from a coding effort.

An example of peeling out the most favored target of an indirect branch with correlated branch history is shown in Example 3-8.

Example 3-8. A Peeling Technique to Reduce Indirect Branch Misprediction

```
function ()
{
   int n = rand();                    // Random integer 0 to RAND_MAX
    if( ! (n & 0x01) ) THEN
          n = 0;                      // n will be 0 half the times
    if (!n) THEN
          handle_0();                 // Peel out the most common target
                                      // with correlated branch history

    {
      switch (n) {
          case 1: handle_1(); break;  // Uncommon
          case 3: handle_3(); break;  // Uncommon

          default: handle_other();    // Make the favored target in
                                      // the fall-through path
          }
    }
}
```

3.4.1.6 Loop Unrolling

Benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows one to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependence chain to expose the critical path.
- Unrolling exposes the code to various other optimizations, such as removal of redundant loads, common subexpression elimination, and so on.

The potential costs of unrolling loops are:

- Unrolling loops whose bodies contain branches increases demand on BTB capacity. If the number of iterations of the unrolled loop is 16 or fewer, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

Assembly/Compiler Coding Rule 14. (H impact, M generality) *Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop.*

Assembly/Compiler Coding Rule 15. (M impact, M generality) *Unroll loops that are frequently executed and have a predictable number of iterations to reduce the number of iterations to 16 or fewer. Do this unless it increases code size so that the working set no longer fits in the instruction cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches).*

Example 3-9 shows how unrolling enables other optimizations.

Example 3-9. Loop Unrolling

```
Before unrolling:
      do i = 1, 100
            if ( i mod 2 == 0 ) then a( i ) = x
                  else a( i ) = y
      enddo
After unrolling
      do i = 1, 100, 2
            a( i ) = y
            a( i+1 ) = x
      enddo
```

In this example, the loop that executes 100 times assigns **X** to every even-numbered element and **Y** to every odd-numbered element. By unrolling the loop you can make assignments more efficiently, removing one branch in the loop body.

3.4.2 FETCH AND DECODE OPTIMIZATION

Intel Core microarchitecture provides several mechanisms to increase front end throughput. Techniques to take advantage of some of these features are discussed below.

3.4.2.1 Optimizing for Microfusion

An Instruction that operates on a register and a memory operand decodes into more micro-ops than its corresponding register-register version. Replacing the equivalent work of the former instruction using the register-register version usually require a sequence of two instructions. The latter sequence is likely to result in reduced fetch bandwidth.

Assembly/Compiler Coding Rule 16. (ML impact, M generality) *For improving fetch/decode throughput, Give preference to memory flavor of an instruction over the register-only flavor of the same instruction, if such instruction can benefit from micro-fusion.*

The following examples are some of the types of micro-fusions that can be handled by all decoders:

- All stores to memory, including store immediate. Stores execute internally as two separate micro-ops: store-address and store-data.
- All “read-modify” (load+op) instructions between register and memory, for example:
  ADDPS XMM9, OWORD PTR [RSP+40]
  FADD DOUBLE PTR [RDI+RSI*8]
  XOR RAX, QWORD PTR [RBP+32]
- All instructions of the form “load and jump,” for example:
  JMP [RDI+200]
  RET
- CMP and TEST with immediate operand and memory.

An Intel 64 instruction with RIP relative addressing is not micro-fused in the following cases:

- When an additional immediate is needed, for example:
  CMP [RIP+400], 27
  MOV [RIP+3000], 142

- When an RIP is needed for control flow purposes, for example:
  JMP [RIP+5000000]

In these cases, Intel Core microarchitecture and Sandy Bridge microarchitecture provide a 2 micro-op flow from decoder 0, resulting in a slight loss of decode bandwidth since 2 micro-op flow must be steered to decoder 0 from the decoder with which it was aligned.

RIP addressing may be common in accessing global data. Since it will not benefit from micro-fusion, compiler may consider accessing global data with other means of memory addressing.

3.4.2.2 Optimizing for Macrofusion

Macrofusion merges two instructions to a single micro-op. Intel Core microarchitecture performs this hardware optimization under limited circumstances.

The first instruction of the macro-fused pair must be a CMP or TEST instruction. This instruction can be REG-REG, REG-IMM, or a micro-fused REG-MEM comparison. The second instruction (adjacent in the instruction stream) should be a conditional branch.

Since these pairs are common ingredient in basic iterative programming sequences, macrofusion improves performance even on un-recompiled binaries. All of the decoders can decode one macro-fused pair per cycle, with up to three other instructions, resulting in a peak decode bandwidth of five
instructions per cycle.

Each macro-fused instruction executes with a single dispatch. This process reduces latency, which in this case shows up as a cycle removed from branch mispredict penalty. Software also gain all other fusion benefits: increased rename and retire bandwidth, more storage for instructions in-flight, and power savings from representing more work in fewer bits.

The following list details when you can use macrofusion:

- CMP or TEST can be fused when comparing:
  - REG-REG. For example: **CMP EAX,ECX**; **JZ** label
  - REG-IMM. For example: **CMP EAX**,0x80; **JZ** label
  - REG-MEM. For example: **CMP EAX**,[**ECX**]; **JZ** label
  - MEM-REG. For example: **CMP [EAX],ECX**; **JZ** label
- TEST can fused with all conditional jumps.

CMP can be fused with only the following conditional jumps in Intel Core microarchitecture. These conditional jumps check carry flag (CF) or zero flag (ZF). jump.

The list of macrofusion-capable conditional jumps are:

- JA OR JNBE
- JAE OR JNB OR JNC
- JE OR JZ
- JNA OR JBE
- JNAE OR JC OR JB
- JNE OR JNZ

CMP and TEST can not be fused when comparing MEM-IMM (e.g. **CMP [EAX]**,0x80; **JZ** label). Macrofusion is not supported in 64-bit mode for Intel Core microarchitecture.

Nehalem microarchitecture supports the following enhancements in macrofusion:

- CMP can be fused with the following conditional jumps (that was not supported in Intel Core microarchitecture):
  - JL or JNGE

  - JGE or JNL
  - JLE or JNG
  - JG or JNLE
- Macrofusion is supported in 64-bit mode.

Enhanced macrofusion support in Sandy Bridge microarchitecture is summarized in Table 3-1 with additional information in Example 3-14:

Table 3-1. Macro-Fusible Instructions in the Sandy Bridge Microarchitecture

Instructions	TEST	AND	CMP	ADD	SUB	INC	DEC
JO/JNO	Y	Y	N	N	N	N	N
JC/JB/JAE/JNB	Y	Y	Y	Y	Y	N	N
JE/JZ/JNE/JNZ	Y	Y	Y	Y	Y	Y	Y
JNA/JBE/JA/JNBE	Y	Y	Y	Y	Y	N	N
JS/JNS/JP/JPE/JNP/JPO	Y	Y	N	N	N	N	N
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	Y	Y	Y	Y	Y	Y	Y

- Enhanced macrofusion support in Haswell microarchitecture is summarized in Table 3-2. Macrofusion is supported CMP/TEST/OP with reg-imm, reg-mem, and reg-reg addressing but not mem-imm addressing.

Table 3-2. Macro-Fusible Instructions in the Haswell Microarchitecture

Opcode		JCC	ADD / SUB / CMP	INC / DEC	TEST / AND
70	0F 80	Jo	N	N	Y
71	0F 81	Jno	N	N	Y
72	0F 82	Jc / Jb	Y	N	Y
73	0F 83	Jae / Jnb	Y	N	Y
74	0F 84	Je / Jz	Y	Y	Y
75	0F 85	Jne / Jnz	Y	Y	Y
76	0F 86	Jna / Jbe	Y	N	Y
77	0F 87	Ja / Jnbe	Y	N	Y
78	0F 88	Js	N	N	Y
79	0F 89	Jns	N	N	Y
7A	0F 8A	Jp / Jpe	N	N	Y
7B	0F 8B	Jnp / Jpo	N	N	Y
7C	0F 8C	Jl / Jnge	Y	Y	Y
7D	0F 8D	Jge / Jnl	Y	Y	Y

Table 3-2. Macro-Fusible Instructions in the Haswell Microarchitecture (Contd.)

Opcode		JCC	ADD / SUB / CMP	INC / DEC	TEST / AND
7E	0F 8E	Jle / Jng	Y	Y	Y
7F	0F 8F	Jg / Jnle	Y	Y	Y

Assembly/Compiler Coding Rule 17. (M impact, ML generality) *Employ macrofusion where possible using instruction pairs that support macrofusion. Prefer TEST over CMP if possible. Use unsigned variables and unsigned jumps when possible. Try to logically verify that a variable is non-negative at the time of comparison. Avoid CMP or TEST of MEM-IMM flavor when possible. However, do not add other instructions to avoid using the MEM-IMM flavor.*

Example 3-10. Macrofusion, Unsigned Iteration Count

	Without Macrofusion	With Macrofusion

C code	for (int[1] i = 0; i < 1000; i++)
	  a++;
		for (unsigned int[2] i = 0; i < 1000; i++)
		  a++;

Disassembly	for (int i = 0; i < 1000; i++)
	mov dword ptr [i], 0
	jmp First
	Loop:
	mov eax, dword ptr [i]
	add eax, 1
	mov dword ptr [i], eax
	
	First:
	cmp dword ptr [i], 3E8H[3]
	jge End
	  a++;
	mov eax, dword ptr [a]
	addqq eax,1
	mov dword ptr [a], eax
	jmp Loop
	End:
		for (unsigned int i = 0; i < 1000; i++)
		xor eax, eax
		mov dword ptr [i], eax
		jmp First
		Loop:
		mov eax, dword ptr [i]
		add eax, 1
		mov dword ptr [i], eax
		
		First:
		cmp eax, 3E8H [4]
		jae End
		  a++;
		mov eax, dword ptr [a]
		add eax, 1
		mov dword ptr [a], eax
		jmp Loop
		End:

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.
4. CMP REG-IMM, JAE permits macrofusion.

Example 3-11. Macrofusion, If Statement

	Without Macrofusion	With Macrofusion

C code	int[1] a = 7;
	if (a < 77)
	  a++;
	else
	  a--;
		unsigned int[2] a = 7;
		if (a < 77)
		  a++;
		else
		  a--;

Disassembly	int a = 7;
	mov dword ptr [a], 7
	if (a < 77)
	cmp dword ptr [a], 4DH [3]
	jge Dec
		unsigned int a = 7;
		mov dword ptr [a], 7
		if (a < 77)
		mov eax, dword ptr [a]
		cmp eax, 4DH
		jae Dec

	a++;
	mov eax, dword ptr [a]
	add eax, 1
	mov dword ptr [a], eax
	else
	jmp End
	  a--;
	Dec:
	mov eax, dword ptr [a]
	sub eax, 1
	mov dword ptr [a], eax
	End::
		a++;
		add eax,1
		mov dword ptr [a], eax
		else
		jmp End
		  a--;
		Dec:
		sub eax, 1
		mov dword ptr [a], eax
		End::

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.

Assembly/Compiler Coding Rule 18. (M impact, ML generality) *Software can enable macro fusion when it can be logically determined that a variable is non-negative at the time of comparison; use TEST appropriately to enable macrofusion when comparing a variable with 0.*

Example 3-12. Macrofusion, Signed Variable

Without Macrofusion	With Macrofusion
test ecx, ecx jle OutSideTheIF cmp ecx, 64H jge OutSideTheIF <IF BLOCK CODE> OutSideTheIF:	test ecx, ecx jle OutSideTheIF cmp ecx, 64H jae OutSideTheIF <IF BLOCK CODE> OutSideTheIF:

For either signed or unsigned variable ‘a’; “CMP a,0” and “TEST a,a” produce the same result as far as the flags are concerned. Since TEST can be macro-fused more often, software can use “TEST a,a” to replace “CMP a,0” for the purpose of enabling macrofusion.

Example 3-13. Macrofusion, Signed Comparison

C Code	Without Macrofusion	With Macrofusion
if (a == 0)	cmp a, 0 jne lbl ... lbl:	test a, a jne lbl ... lbl:
if (a >= 0)	cmp a, 0 jl lbl; ... lbl:	test a, a jl lbl ... lbl:

The Sandy Bridge microarchitecture enables more arithmetic and logic instructions to macro-fuse with conditional branches. In loops where the ALU ports are already congested, performing one of these macrofusions can relieve the pressure, as the macro-fused instruction consumes only port 5, instead of an ALU port plus port 5.

In Example 3-14, the “add/cmp/jnz” loop contains two ALU instructions that can be dispatched via either port 0, 1, 5. So there is higher probability of port 5 might bind to either ALU instruction causing JNZ to wait a cycle. The “sub/jnz” loop, the likelihood of ADD/SUB/JNZ can be dispatched in the same cycle is increased because only SUB is free to bind with either port 0, 1, 5.

Example 3-14. Additional Macrofusion Benefit in the Sandy Bridge Microarchitecture

Add + cmp + jnz alternative		Loop control with sub + jnz	
lea	rdx, buff	lea	rdx, buff - 4
xor	rcx, rcx	xor	rcx, LEN
xor	eax, eax	xor	eax, eax
loop:		loop:	
add	eax, [rdx + 4 * rcx]	add	eax, [rdx + 4 * rcx]
add	rcx, 1	sub	rcx, 1
cmp	rcx, LEN	jnz	loop
jnz	loop		

3.4.2.3 Length-Changing Prefixes (LCP)

The length of an instruction can be up to 15 bytes in length. Some prefixes can dynamically change the length of an instruction that the decoder must recognize. Typically, the pre-decode unit will estimate the length of an instruction in the byte stream assuming the absence of LCP. When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle. Normal queuing throughout of the machine pipeline generally cannot hide LCP penalties.

The prefixes that can dynamically change the length of a instruction include:

- Operand size prefix (0x66).
- Address size prefix (0x67).

The instruction MOV DX, 01234h is subject to LCP stalls in processors based on Intel Core microarchitecture, and in Intel Core Duo and Intel Core Solo processors. Instructions that contain imm16 as part of their fixed encoding but do not require LCP to change the immediate size are not subject to LCP stalls. The REX prefix (4xh) in 64-bit mode can change the size of two classes of instruction, but does not cause an LCP penalty.

If the LCP stall happens in a tight loop, it can cause significant performance degradation. When decoding is not a bottleneck, as in floating-point heavy code, isolated LCP stalls usually do not cause performance degradation.

Assembly/Compiler Coding Rule 19. (MH impact, MH generality) *Favor generating code using imm8 or imm32 values instead of imm16 values.*

If imm16 is needed, load equivalent imm32 into a register and use the word value in the register instead.

Double LCP Stalls

Instructions that are subject to LCP stalls and cross a 16-byte fetch line boundary can cause the LCP stall to trigger twice. The following alignment situations can cause LCP stalls to trigger twice:

- An instruction is encoded with a MODR/M and SIB byte, and the fetch line boundary crossing is between the MODR/M and the SIB bytes.
- An instruction starts at offset 13 of a fetch line references a memory location using register and immediate byte offset addressing mode.

The first stall is for the 1st fetch line, and the 2nd stall is for the 2nd fetch line. A double LCP stall causes a decode penalty of 11 cycles.

The following examples cause LCP stall once, regardless of their fetch-line location of the first byte of the instruction:

```
ADD DX, 01234H
ADD word ptr [EDX], 01234H
ADD word ptr 012345678H[EDX], 01234H
ADD word ptr [012345678H], 01234H
```

The following instructions cause a double LCP stall when starting at offset 13 of a fetch line:

```
ADD word ptr [EDX+ESI], 01234H
ADD word ptr 012H[EDX], 01234H
ADD word ptr 012345678H[EDX+ESI], 01234H
```

To avoid double LCP stalls, do not use instructions subject to LCP stalls that use SIB byte encoding or addressing mode with byte displacement.

False LCP Stalls

False LCP stalls have the same characteristics as LCP stalls, but occur on instructions that do not have any imm16 value.

False LCP stalls occur when (a) instructions with LCP that are encoded using the F7 opcodes, and (b) are located at offset 14 of a fetch line. These instructions are: not, neg, div, idiv, mul, and imul. False LCP experiences delay because the instruction length decoder can not determine the length of the instruction before the next fetch line, which holds the exact opcode of the instruction in its MODR/M byte.

The following techniques can help avoid false LCP stalls:

- Upcast all short operations from the F7 group of instructions to long, using the full 32 bit version.
- Ensure that the F7 opcode never starts at offset 14 of a fetch line.

Assembly/Compiler Coding Rule 20. (M impact, ML generality) *Ensure instructions using 0xF7 opcode byte does not start at offset 14 of a fetch line; and avoid using these instruction to operate on 16-bit data, upcast short data to 32 bits.*

Example 3-15. Avoiding False LCP Delays with 0xF7 Group Instructions

A Sequence Causing Delay in the Decoder	Alternate Sequence to Avoid Delay
neg word ptr a	movsx eax, word ptr a neg eax mov word ptr a, AX

3.4.2.4 Optimizing the Loop Stream Detector (LSD)

The LSD detects loops that have many iterations and fit into the µop-queue. The µop-queue streams the loop until a branch miss-prediction inevitably ends it.

LSD improves fetch bandwidth. In single thread mode, it saves power by allowing the front-end to sleep. In multi-thread mode, front-resource can better serve the other thread.

Loops qualify for LSD replay if all the following conditions are met:

- Loop body size up to 60 µops, with up to 15 taken branches, and up to 15 64-byte fetch lines.
- No CALL or RET.
- No mismatched stack operations (e.g., more PUSH than POP).
- More than ~20 iterations.

Many calculation-intensive loops, searches, and software string moves match these characteristics. These loops exceed the BPU prediction capacity and always terminate in a branch misprediction.

Assembly/Compiler Coding Rule 21. (MH impact, MH generality) *Break up a loop body with a long sequence of instructions into loops of shorter instruction blocks of no more than the size of the LSD.*

Allocation bandwidth in Ice Lake Client microarchitecture increased from 4 µops per cycle to 5 µops per cycle.

Assume a loop that qualifies for LSD has 23 µops in the loop body. The hardware unrolls the loop so it still fits into the µop-queue, in this case twice. The loop in the µop-queue thus takes 46 µops.

The loop is sent to allocation 5 µops per cycle. After 45 out of the 46 µops are sent, in the next cycle only a single µop is sent, which means that in that cycle, 4 of the allocation slots are wasted. This pattern repeats itself, until the loop is exited by a misprediction. Hardware loop unrolling minimizes the number of wasted slots during LSD.

3.4.2.5 Optimization for Decoded ICache

The decoded ICache is a new feature in Sandy Bridge microarchitecture. Running the code from the Decoded ICache has two advantages:

- Higher bandwidth of micro-ops feeding the out-of-order engine.
- The front end does not need to decode the code that is in the Decoded ICache; this saves power.

There is overhead in switching between the Decoded ICache and the legacy decode pipeline. If your code switches frequently between the front end and the Decoded ICache, the penalty may be higher than running only from the legacy pipeline.

To ensure “hot” code is feeding from the decoded ICache:

- Make sure each hot code block is less than about 750 instructions. Specifically, do not unroll to more than 750 instructions in a loop. This should enable Decoded ICache residency even when hyper-threading is enabled.
- For applications with very large blocks of calculations inside a loop, consider loop-fission: split the loop into multiple loops that fit in the Decoded ICache, rather than a single loop that overflows.
- If an application can be sure to run with only one thread per core, it can increase hot code block size to about 1500 instructions.

Dense Read-Modify-Write Code

The Decoded ICache can hold only up to 18 micro-ops per each 32 byte aligned memory chunk. Therefore, code with a high concentration of instructions that are encoded in a small number of bytes, yet have many micro-ops, may overflow the 18 micro-op limitation and not enter the Decoded ICache. Read-modify-write (RMW) instructions are a good example of such instructions.

RMW instructions accept one memory source operand, one register source operand, and use the source memory operand as the destination. The same functionality can be achieved by two or three instructions: the first reads the memory source operand, the second performs the operation with the second register source operand, and the last writes the result back to memory. These instructions usually result in the same number of micro-ops but use more bytes to encode the same functionality.

One case where RMW instructions may be used extensively is when the compiler optimizes aggressively for code size.

Here are some possible solutions to fit the hot code in the Decoded ICache:

- Replace RMW instructions with two or three instructions that have the same functionality. For example, “adc [rdi], rcx“ is only three bytes long; the equivalent sequence “adc rax, [rdi]“ + “mov [rdi], rax“ has a footprint of six bytes.
- Align the code so that the dense part is broken down among two different 32-byte chunks. This solution is useful when using a tool that aligns code automatically, and is indifferent to code changes.
- Spread the code by adding multiple byte NOPs in the loop. Note that this solution adds micro-ops for execution.

Align Unconditional Branches for Decoded ICache

For code entering the Decoded ICache, each unconditional branch is the last micro-op occupying a Decoded ICache Way. Therefore, only three unconditional branches per a 32 byte aligned chunk can enter the Decoded ICache.

Unconditional branches are frequent in jump tables and switch declarations. Below are examples for these constructs, and methods for writing them so that they fit in the Decoded ICache.

Compilers create jump tables for C++ virtual class methods or DLL dispatch tables. Each unconditional branch consumes five bytes; therefore up to seven of them can be associated with a 32-byte chunk. Thus jump tables may not fit in the Decoded ICache if the unconditional branches are too dense in each 32Byte-aligned chunk. This can cause performance degradation for code executing before and after the branch table.

The solution is to add multi-byte NOP instructions among the branches in the branch table. This may increases code size and should be used cautiously. However, these NOPs are not executed and therefore have no penalty in later pipe stages.

Switch-Case constructs represents a similar situation. Each evaluation of a case condition results in an unconditional branch. The same solution of using multi-byte NOP can apply for every three consecutive unconditional branches that fits inside an aligned 32-byte chunk.

Two Branches in a Decoded ICache Way

The Decoded ICache can hold up to two branches in a way. Dense branches in a 32 byte aligned chunk, or their ordering with other instructions may prohibit all the micro-ops of the instructions in the chunk from entering the

Decoded ICache. This does not happen often. When it does happen, you can space the code with NOP instructions where appropriate. Make sure that these NOP instructions are not part of hot code.

Assembly/Compiler Coding Rule 22. (M impact, M generality) *Avoid putting explicit references to* ESP *in a sequence of stack operations (*POP, PUSH, CALL, RET*).*

3.4.2.6 Other Decoding Guidelines

Assembly/Compiler Coding Rule 23. (ML impact, L generality) *Use simple instructions that are less than eight bytes in length.*

Assembly/Compiler Coding Rule 24. (M impact, MH generality) *Avoid using prefixes to change the size of immediate and displacement.*

Long instructions (more than seven bytes) may limit the number of decoded instructions per cycle. Each prefix adds one byte to the length of instruction, possibly limiting the decoder's throughput. In addition, multiple prefixes can only be decoded by the first decoder. These prefixes also incur a delay when decoded. If multiple prefixes or a prefix that changes the size of an immediate or displacement cannot be avoided, schedule them behind instructions that stall the pipe for some other reason.

3.5 OPTIMIZING THE EXECUTION CORE

The superscalar, out-of-order execution core(s) in recent generations of microarchitectures contain multiple execution hardware resources that can execute multiple micro-ops in parallel. These resources generally ensure that micro-ops execute efficiently and proceed with fixed latencies. General guidelines to make use of the available parallelism are:

- Follow the rules (see Section 3.4) to maximize useful decode bandwidth and front end throughput. These rules include favoring single micro-op instructions and taking advantage of micro-fusion, Stack pointer tracker and macrofusion.
- Maximize rename bandwidth. Guidelines are discussed in this section and include properly dealing with partial registers, ROB read ports and instructions which causes side-effects on flags.
- Scheduling recommendations on sequences of instructions so that multiple dependency chains are alive in the reservation station (RS) simultaneously, thus ensuring that your code utilizes maximum parallelism.
- Avoid hazards, minimize delays that may occur in the execution core, allowing the dispatched micro-ops to make progress and be ready for retirement quickly.

3.5.1 INSTRUCTION SELECTION

Some execution units are not pipelined, this means that micro-ops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle.

It is generally a good starting point to select instructions by considering the number of micro-ops associated with each instruction, favoring in the order of: single micro-op instructions, simple instruction with less than 4 micro-ops, and last instruction requiring microsequencer ROM (micro-ops which are executed out of the microsequencer involve extra overhead).

Assembly/Compiler Coding Rule 25. (M impact, H generality) *Favor single-micro-operation instructions. Also favor instruction with shorter latencies.*

A compiler may be already doing a good job on instruction selection. If so, user intervention usually is not necessary.

Assembly/Compiler Coding Rule 26. (M impact, L generality) *Avoid prefixes, especially multiple non-0F-prefixed opcodes.*

Assembly/Compiler Coding Rule 27. (M impact, L generality) *Do not use many segment registers.*

Assembly/Compiler Coding Rule 28. (M impact, M generality) *Avoid using complex instructions (for example, enter, leave, or loop) that have more than four µops and require multiple cycles to decode. Use sequences of simple instructions instead.*

Assembly/Compiler Coding Rule 29. (MH impact, M generality) *Use push/pop to manage stack space and address adjustments between function calls/returns instead of enter/leave. Using enter instruction with non-zero immediates can experience significant delays in the pipeline in addition to misprediction.*

Theoretically, arranging instructions sequence to match the 4-1-1-1 template applies to processors based on Intel Core microarchitecture. However, with macrofusion and micro-fusion capabilities in the front end, attempts to schedule instruction sequences using the 4-1-1-1 template will likely provide diminishing returns.

Instead, software should follow these additional decoder guidelines:

- If you need to use multiple micro-op, non-microsequenced instructions, try to separate by a few single micro-op instructions. The following instructions are examples of multiple micro-op instruction not requiring micro-sequencer:

  ADC/SBB
  CMOVcc
  Read-modify-write instructions

- If a series of multiple micro-op instructions cannot be separated, try breaking the series into a different equivalent instruction sequence. For example, a series of read-modify-write instructions may go faster if sequenced as a series of read-modify + store instructions. This strategy could improve performance even if the new code sequence is larger than the original one.

3.5.1.1 Integer Divide

Typically, an integer divide is preceded by a CWD or **CDQ** instruction. Depending on the operand size, divide instructions use **DX:AX** or EDX:EAX for the dividend. The **CWD** or **CDQ** instructions sign-extend AX or EAX into DX or EDX, respectively. These instructions have denser encoding than a shift and move would be, but they generate the same number of micro-ops. If AX or EAX is known to be positive, replace these instructions with:

```
xor dx, dx
```

or

```
xor edx, edx
```

Modern compilers typically can transform high-level language expression involving integer division where the divisor is a known integer constant at compile time into a faster sequence using IMUL instruction instead. Thus programmers should minimize integer division expression with divisor whose value can not be known at compile time.

Alternately, if certain known divisor value are favored over other unknown ranges, software may consider isolating the few favored, known divisor value into constant-divisor expressions.

Section 13.2.4 describes more detail of using MUL/IMUL to replace integer divisions.

3.5.1.2 Using LEA

In Sandy Bridge microarchitecture, there are two significant changes to the performance characteristics of LEA instruction:

- LEA can be dispatched via port 1 and 5 in most cases, doubling the throughput over prior generations. However this apply only to LEA instructions with one or two source operands.

Example 3-16. Independent Two-Operand LEA Example

```
    mov     edx, N
    mov     eax, X
    mov     ecx, Y

loop:
    lea     ecx, [ecx + ecx]        // ecx = ecx*2
    lea     eax, [eax + eax *4]     // eax = eax*5
    and     ecx, 0xff
    and     eax, 0xff
    dec     edx
    jg      loop
```

- For LEA instructions with three source operands and some specific situations, instruction latency has increased to 3 cycles, and must dispatch via port 1:
  - LEA that has all three source operands: base, index, and offset.
  - LEA that uses base and index registers where the base is EBP, RBP, or R13.
  - LEA that uses RIP relative addressing mode.
  - LEA that uses 16-bit addressing mode.

Example 3-17. Alternative to Three-Operand LEA

Three operand LEA is slower	Two-operand LEA alternative	Alternative Two

#define K 1
uint32 an = 0;
uint32 N= mi_N;
mov ecx, N
xor esi, esi;
xor edx, edx;
cmp ecx, 2;
jb finished;
dec ecx;
 	
loop1:
mov edi, esi;
lea esi, [K+esi+edx];
and esi, 0xFF;
mov edx, edi;
dec ecx;
jnz loop1;
finished:
mov [an] ,esi;
	#define K 1
	uint32 an = 0;
	uint32 N= mi_N;
	mov ecx, N
	xor esi, esi;
	xor edx, edx;
	cmp ecx, 2;
	jb finished;
	dec ecx;
	
	loop1:
	mov edi, esi;
	lea esi, [K+edx];
	lea esi, [esi+edx];
	and esi, 0xFF;
	mov edx, edi;
	dec ecx;
	jnz loop1;
	finished:
	mov [an] ,esi;
		#define K 1
		uint32 an = 0;
		uint32 N= mi_N;
		mov ecx, N
		xor esi, esi;
		mov edx, K;
		cmp ecx, 2;
		jb finished;
		mov eax, 2
		dec ecx;
		
		loop1:
		mov edi, esi;
		lea esi, [esi+edx];
		and esi, 0xFF;
		lea edx, [edi +K];
		dec ecx;
		jnz loop1;
		finished:
		mov [an] ,esi;

The LEA instruction or a sequence of LEA, ADD, SUB and SHIFT instructions can replace constant multiply instructions. The LEA instruction can also be used as a multiple operand addition instruction, for example:

LEA ECX, [EAX + EBX*4 + A]

Using LEA in this way may avoid register usage by not tying up registers for operands of arithmetic instructions. This use may also save code space.

If the LEA instruction uses a shift by a constant amount then the latency of the sequence of µops is shorter if adds are used instead of a shift, and the LEA instruction may be replaced with an appropriate sequence of µops. This, however, increases the total number of µops, leading to a trade-off.

Assembly/Compiler Coding Rule 30. (ML impact, L generality) *If an LEA instruction using the scaled index is on the critical path, a sequence with ADDs may be better.*

3.5.1.3 ADC and SBB in Sandy Bridge Microarchitecture

The throughput of ADC and SBB in Sandy Bridge microarchitecture is one cycle, compared to 1.5 to two cycles in the prior generation. These two instructions are useful in numeric handling of integer data types that are wider than the maximum width of native hardware.

Example 3-18. Examples of 512-bit Additions

```
//Add 64-bit to 512 Number
    lea     rsi, gLongCounter
    lea     rdi, gStepValue
    mov     rax, [rdi]
    xor     rcx, rcx
loop_start:
    mov     r10, [rsi+rcx]
    add     r10, rax
    mov     [rsi+rcx], r10

    mov     r10, [rsi+rcx+8]
    adc     r10, 0
    mov     [rsi+rcx+8], r10

    mov     r10, [rsi+rcx+16]
    adc     r10, 0
    mov     [rsi+rcx+16], r10
    mov     r10, [rsi+rcx+24]
    adc     r10, 0
    mov     [rsi+rcx+24], r10

    mov     r10, [rsi+rcx+32]
    adc     r10, 0
    mov     [rsi+rcx+32], r10
    mov     r10, [rsi+rcx+40]
    adc     r10, 0
    mov     [rsi+rcx+40], r10
```

```
// 512-bit Addition
loop1:
    mov     rax, [StepValue]
    add     rax, [LongCounter]
    mov     LongCounter, rax
    mov     rax, [StepValue+8]
    adc     rax, [LongCounter+8]
    mov     LongCounter+8, rax
    mov     rax, [StepValue+16]
    adc     rax, [LongCounter+16]

    mov     LongCounter+16, rax
    mov     rax, [StepValue+24]
    adc     rax, [LongCounter+24]

    mov     LongCounter+24, rax
    mov     rax, [StepValue+32]
    adc     rax, [LongCounter+32]

    mov     LongCounter+32, rax
    mov     rax, [StepValue+40]
    adc     rax, [LongCounter+40]

    mov     LongCounter+40, rax
    mov     rax, [StepValue+48]
    adc     rax, [LongCounter+48]
```

Example 3-18. Examples of 512-bit Additions (Contd.)

```
mov     r10, [rsi+rcx+48]
adc     r10, 0
mov     [rsi+rcx+48], r10

mov     r10, [rsi+rcx+56]
adc     r10, 0
mov     [rsi+rcx+56], r10
add     rcx, 64
cmp     rcx, SIZE
jnz     loop_start
```

```
mov     LongCounter+48, rax
mov     rax, [StepValue+56]
adc     rax, [LongCounter+56]

mov     LongCounter+56, rax
dec     rcx
jnz     loop1
```

3.5.1.4 Bitwise Rotation

Bitwise rotation can choose between rotate with count specified in the CL register, an immediate constant and by 1 bit. Generally, The rotate by immediate and rotate by register instructions are slower than rotate by 1 bit. The rotate by 1 instruction has the same latency as a shift.

Assembly/Compiler Coding Rule 31. (ML impact, L generality) *Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction.*

In Sandy Bridge microarchitecture, ROL/ROR by immediate has 1-cycle throughput, SHLD/SHRD using the same register as source and destination by an immediate constant has 1-cycle latency with 0.5 cycle throughput. The "ROL/ROR reg, imm8" instruction has two micro-ops with the latency of 1-cycle for the rotate register result and 2-cycles for the flags, if used.

In Ivy Bridge microarchitecture, The "ROL/ROR reg, imm8" instruction with immediate greater than 1, is one micro-op with one-cycle latency when the overflow flag result is used. When the immediate is one, dependency on the overflow flag result of ROL/ROR by a subsequent instruction will see the ROL/ROR instruction with two-cycle latency.

3.5.1.5 Variable Bit Count Rotation and Shift

In Sandy Bridge microarchitecture, The "ROL/ROR/SHL/SHR reg, cl" instruction has three micro-ops. When the flag result is not needed, one of these micro-ops may be discarded, providing better performance in many common usages. When these instructions update partial flag results that are subsequently used, the full three micro-ops flow must go through the execution and retirement pipeline, experiencing slower performance. In Ivy Bridge microarchitecture, executing the full three micro-ops flow to use the updated partial flag result has additional delay. Consider the looped sequence below:

```
loop:
        shl eax, cl
        add ebx, eax
        dec edx ; DEC does not update carry, causing SHL to execute slower three micro-ops flow
        jnz loop
```

The DEC instruction does not modify the carry flag. Consequently, the SHL EAX, CL instruction needs to execute the three micro-ops flow in subsequent iterations. The SUB instruction will update all flags. So replacing DEC with SUB will allow SHL EAX, CL to execute the two micro-ops flow.

3.5.1.6 Address Calculations

For computing addresses, use the addressing modes rather than general-purpose computations. Internally, memory reference instructions can have four operands:

- Relocatable load-time constant.
- Immediate constant.
- Base register.
- Scaled index register.

Note that the latency and throughput of LEA with more than two operands are slower in Sandy Bridge microarchitecture (see Section 3.5.1.2). Addressing modes that uses both base and index registers will consume more read port resource in the execution engine and may experience more stalls due to availability of read port resources. Software should take care by selecting the speedy version of address calculation.

In the segmented model, a segment register may constitute an additional operand in the linear address calculation. In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

3.5.1.7 Clearing Registers and Dependency Breaking Idioms

Code sequences that modifies partial register can experience some delay in its dependency chain, but can be avoided by using dependency breaking idioms.

In processors based on Intel Core microarchitecture, a number of instructions can help clear execution dependency when software uses these instruction to clear register content to zero. The instructions include:

XOR REG, REG
SUB REG, REG
XORPS/PD XMMREG, XMMREG
PXOR XMMREG, XMMREG
SUBPS/PD XMMREG, XMMREG
PSUBB/W/D/Q XMMREG, XMMREG

In processors based on Sandy Bridge microarchitecture, the instruction listed above plus equivalent AVX counter parts are also zero idioms that can be used to break dependency chains. Furthermore, they do not consume an issue port or an execution unit. So using zero idioms are preferable than moving 0's into the register. The AVX equivalent zero idioms are:

VXORPS/PD XMMREG, XMMREG
VXORPS/PD YMMREG, YMMREG
VPXOR XMMREG, XMMREG
VSUBPS/PD XMMREG, XMMREG
VSUBPS/PD YMMREG, YMMREG
VPSUBB/W/D/Q XMMREG, XMMREG

Microarchitectures that support Intel AVX-512 have the equivalent of zero idioms for the 512-bit registers using the unmasked versions of the instructions:

VXORPS/PD ZMMREG, ZMMREG
VPXOR ZMMREG, ZMMREG
VSUBPS/PD ZMMREG, ZMMREG
VPSUBB/W/D/Q ZMMREG, ZMMREG

The **XOR** and **SUB** instructions can be used to clear execution dependencies on the zero evaluation of the destination register.

Assembly/Compiler Coding Rule 32. (M impact, ML generality) *Use dependency-breaking-idiom instructions to set a register to* **0***, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move* **0** *into the register instead. This requires more code space than using* **XOR** *and* **SUB***, but avoids setting the condition codes.*

Example 3-19 of using pxor to break dependency idiom on a XMM register when performing negation on the elements of an array.

```
int a[4096], b[4096], c[4096];
For ( int i = 0; i < 4096; i++ )
        C[i] = - ( a[i] + b[i] );
```

Example 3-19. Clearing Register to Break Dependency While Negating Array Elements

Negation (-x = (x XOR (-1)) - (-1) without breaking dependency		Negation (-x = 0 -x) using PXOR reg, reg breaks dependency	
lea	eax, a	lea	eax, a
lea	ecx, b	lea	ecx, b
lea	edi, c	lea	edi, c
xor	edx, edx	xor	edx, edx
movdqa	xmm7, allone	lp:	
lp:			
movdqa	xmm0, [eax + edx]	movdqa	xmm0, [eax + edx]
paddd	xmm0, [ecx + edx]	paddd	xmm0, [ecx + edx]
pxor	xmm0, xmm7	pxor	xmm7, xmm7
psubd	xmm0, xmm7	psubd	xmm7, xmm0
movdqa	[edi + edx], xmm0	movdqa	[edi + edx], xmm7
add	edx, 16	add	edx,16
cmp	edx, 4096	cmp	edx, 4096
jl	lp	jl	lp

Assembly/Compiler Coding Rule 33. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using MOVZX.*

Sometimes sign-extended semantics can be maintained by zero-extending operands. For example, the C code in the following statements does not need sign extension, nor does it need prefixes for operand size overrides:

```
static short INT a, b;
IF (a == b) {
  . . .
}
```

Code for comparing these 16-bit operands might be:

```
MOVZW EAX, [a]
MOVZW EBX, [b]
CMP   EAX, EBX
```

These circumstances tend to be common. However, the technique will not work if the compare is for greater than, less than, greater than or equal, and so on, or if the values in eax or ebx are to be used in another operation where sign extension is required.

Assembly/Compiler Coding Rule 34. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.*

The trace cache can be packed more tightly when instructions with operands that can only be represented as 32 bits are not adjacent.

Assembly/Compiler Coding Rule 35. (ML impact, L generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as sign-extended 16-bit immediates near each other. Try to schedule µops that have no immediate immediately before or after µops with 32-bit immediates.*

3.5.1.8 Compares

Use **TEST** when comparing a value in a register with zero. **TEST** essentially **AND**s operands together without writing to a destination register. **TEST** is preferred over **AND** because **AND** produces an extra result register. **TEST** is better than **CMP ..., 0** because the instruction size is smaller.

Use **TEST** when comparing the result of a logical **AND** with an immediate constant for equality or inequality if the register is **EAX** for cases such as:

```
IF (AVAR & 8) { }
```

The **TEST** instruction can also be used to detect rollover of modulo of a power of 2. For example, the C code:

```
IF ( (AVAR % 16) == 0 ) { }
```

can be implemented using:

```
TEST    EAX, 0x0F
JNZ     AfterIf
```

Using the **TEST** instruction between the instruction that may modify part of the flag register and the instruction that uses the flag register can also help prevent partial flag register stall.

Assembly/Compiler Coding Rule 36. (ML impact, M generality) *Use the* **TEST** *instruction instead of* **AND** *when the result of the logical* **AND** *is not used. This saves µops in execution. Use a* **TEST** *of a register with itself instead of a* **CMP** *of the register to zero, this saves the need to encode the zero and saves encoding space. Avoid comparing a constant to a memory operand. It is preferable to load the memory operand and compare the constant to a register.*

Often a produced value must be compared with zero, and then used in a branch. Because most Intel architecture instructions set the condition codes as part of their execution, the compare instruction may be eliminated. Thus the operation can be tested directly by a **JCC** instruction. The notable exceptions are **MOV** and **LEA**. In these cases, use **TEST**.

Assembly/Compiler Coding Rule 37. (ML impact, M generality) *Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a* **TEST** *instruction instead of a compare. Be certain that any code transformations made do not introduce problems with overflow.*

3.5.1.9 Using NOPs

Code generators generate a no-operation (NOP) to align instructions. Examples of NOPs of different lengths in 32-bit mode are shown in Table 3-3.

Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H

Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction (Contd.)

Length	Assembly	Byte Sequence
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

These are all true NOPs, having no effect on the state of the machine except to advance the EIP. Because NOPs require hardware resources to decode and execute, use the fewest number to achieve the desired padding.

The one byte NOP:[**XCHG EAX**,**EAX**] has special hardware support. Although it still consumes a µop and its accompanying resources, the dependence upon the old value of **EAX** is removed. This µop can be executed at the earliest possible opportunity, reducing the number of outstanding instructions, and is the lowest cost NOP.

The other NOPs have no special hardware support. Their input and output registers are interpreted by the hardware. Therefore, a code generator should arrange to use the register containing the oldest value as input, so that the NOP will dispatch and release RS resources at the earliest possible opportunity.

Try to observe the following NOP generation priority:

- Select the smallest number of NOPs and pseudo-NOPs to provide the desired padding.
- Select NOPs that are least likely to execute on slower execution unit clusters.
- Select the register arguments of NOPs to reduce dependencies.

3.5.1.10 Mixing SIMD Data Types

Previous microarchitectures (before Intel Core microarchitecture) do not have explicit restrictions on mixing integer and floating-point (FP) operations on XMM registers. For Intel Core microarchitecture, mixing integer and floating-point operations on the content of an XMM register can degrade performance. Software should avoid mixed-use of integer/FP operation on XMM registers. Specifically:

- Use SIMD integer operations to feed SIMD integer operations. Use PXOR for idiom.
- Use SIMD floating-point operations to feed SIMD floating-point operations. Use XORPS for idiom.
- When floating-point operations are bitwise equivalent, use PS data type instead of PD data type. MOVAPS and MOVAPD do the same thing, but MOVAPS takes one less byte to encode the instruction.

3.5.1.11 Spill Scheduling

The spill scheduling algorithm used by a code generator will be impacted by the memory subsystem. A spill scheduling algorithm is an algorithm that selects what values to spill to memory when there are too many live values to fit in registers. Consider the code in Example 3-20, where it is necessary to spill either **A**, **B**, or **C**.

Example 3-20. Spill Scheduling Code

```
LOOP
    C := ...
    B := ...
    A := A + ...
```

For modern microarchitectures, using dependence depth information in spill scheduling is even more important than in previous processors. The loop-carried dependence in **A** makes it especially important that **A** not be spilled. Not only would a store/load be placed in the dependence chain, but there would also be a data-not-ready stall of the load, costing further cycles.

Assembly/Compiler Coding Rule 38. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies.*

A possibly counter-intuitive result is that in such a situation it is better to put loop invariants in memory than in registers, since loop invariants never have a load blocked by store data that is not ready.

3.5.1.12 Zero-Latency MOV Instructions

In processors based on Ivy Bridge microarchitecture, a subset of register-to-register move operations are executed in the front end (similar to zero-idioms, see Section 3.5.1.7). This conserves scheduling/execution resources in the out-of-order engine. Most forms of register-to-register MOV instructions can benefit from zero-latency MOV. Example 3-21 list the details of those forms that qualify and a small set that do not.

Example 3-21. Zero-Latency MOV Instructions

MOV instructions latency that can be eliminated	MOV instructions latency that cannot be eliminated

MOV reg32, reg32
MOV reg64, reg64
MOVUPD/MOVAPD xmm, xmm
MOVUPD/MOVAPD ymm, ymm
MOVUPS?MOVAPS xmm, xmm
MOVUPS/MOVAPS ymm, ymm
MOVDQA/MOVDQU xmm, xmm
MOVDQA/MOVDQU ymm, ymm
MOVDQA/MOVDQU zmm, zmm
MOVZX reg32, reg8 (if not AH/BH/CH/DH)
MOVZX reg64, reg8 (if not AH/BH/CH/DH)
	MOV reg8, reg8
	MOV reg16, reg16
	MOVZX reg32, reg8 (if AH/BH/CH/DH)
	MOVZX reg64, reg8 (if AH/BH/CH/DH)
	MOVSX

Example 3-22 shows how to process 8-bit integers using MOVZX to take advantage of zero-latency MOV enhancement. Consider

```
X = (X * 3^N ) MOD 256;
Y = (Y * 3^N ) MOD 256;
```

When “MOD 256” is implemented using the “AND 0xff” technique, its latency is exposed in the result-dependency chain. Using a form of MOVZX on a truncated byte input, it can take advantage of zero-latency MOV enhancement and gain about 45% in speed.

Example 3-22. Byte-Granular Data Computation Technique

Use AND Reg32, 0xff	Use MOVZX

mov rsi, N
mov rax, X
mov rcx, Y
loop:
lea rcx, [rcx+rcx*2]
lea rax, [rax+rax*4]
and rcx, 0xff
and rax, 0xff
 	
lea rcx, [rcx+rcx*2]
lea rax, [rax+rax*4]
and rcx, 0xff
and rax, 0xff
sub rsi, 2
jg loop
	mov rsi, N
	mov rax, X
	mov rcx, Y
	loop:
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	
	lea rdx, [rax+rax*4]
	movzx, rax, dl
	llea rdx, [rax+rax*4]
	movzx, rax, dl
	sub rsi, 2
	jg loop

The effectiveness of coding a dense sequence of instructions to rely on a zero-latency MOV instruction must also consider internal resource constraints in the microarchitecture.

Example 3-23. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions

Needing more internal resource for zero-latency MOVs	Needing less internal resource for zero-latency MOVs

mov rsi, N
mov rax, X
mov rcx, Y
loop:
lea rbx, [rcx+rcx*2]
movzx, rcx, bl
lea rdx, [rax+rax*4]
movzx, rax, dl
lea rbx, [rcx+rcx*2]
movzx, rcx, bl
llea rdx, [rax+rax*4]
movzx, rax, dl
sub rsi, 2
jg loop
	mov rsi, N
	mov rax, X
	mov rcx, Y
	loop:
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	lea rbx, [rcx+rcx*2]
	movzx, rcx, bl
	lea rdx, [rax+rax*4]
	movzx, rax, dl
	llea rdx, [rax+rax*4]
	movzx, rax, dl
	sub rsi, 2
	jg loop

In Example 3-23, RBX/RCX and RDX/RAX are pairs of registers that are shared and continuously overwritten. In the right-hand sequence, registers are overwritten with new results immediately, consuming less internal resources provided by the underlying microarchitecture. As a result, it is about 8% faster than the left-hand sequence where internal resources could only support 50% of the attempt to take advantage of zero-latency MOV instructions.

3.5.2 AVOIDING STALLS IN EXECUTION CORE

Although the design of the execution core is optimized to make common cases executes quickly. A micro-op may encounter various hazards, delays, or stalls while making forward progress from the front end to the ROB and RS. The significant cases are:

- ROB Read Port Stalls.
- Partial Register Reference Stalls.
- Partial Updates to XMM Register Stalls.
- Partial Flag Register Reference Stalls.

3.5.2.1 Writeback Bus Conflicts

The writeback bus inside the execution engine is a common resource needed to facilitate out-of-order execution of micro-ops in flight. When the writeback bus is needed at the same time by two micro-ops executing in the same stack of execution units, the younger micro-op will have to wait for the writeback bus to be available. This situation typically will be more likely for short-latency instructions experience a delay when it might have been otherwise ready for dispatching into the execution engine.

Consider a repeating sequence of independent floating-point ADDs with a single-cycle MOV bound to the same dispatch port. When the MOV finds the dispatch port available, the writeback bus can be occupied by the ADD. This delays the MOV operation.

If this problem is detected, you can sometimes change the instruction selection to use a different dispatch port and reduce the writeback contention.

3.5.2.2 Bypass Between Execution Domains

Floating-point (FP) loads have an extra cycle of latency. Moves between FP and SIMD stacks have another additional cycle of latency.

Example:

```
ADDPS  XMM0, XMM1
PAND  XMM0, XMM3
ADDPS  XMM2, XMM0
```

The overall latency for the above calculation is 9 cycles:

- 3 cycles for each ADDPS instruction.
- 1 cycle for the PAND instruction.
- 1 cycle to bypass between the ADDPS floating-point domain to the PAND integer domain.
- 1 cycle to move the data from the PAND integer to the second floating-point ADDPS domain.

To avoid this penalty, organize code to minimize domain changes. Sometimes bypasses cannot be avoided.

Account for bypass cycles when counting the overall latency of your code. If your calculation is latency-bound, you can execute more instructions in parallel or break dependency chains to reduce total latency.

Code that has many bypass domains and is completely latency-bound may run slower on the Intel Core microarchitecture than it did on previous microarchitectures.

3.5.2.3 Partial Register Stalls

Beginning with the Skylake microarchitecture, Partial Register Stalls are no longer treated using micro-operation (UOP) insertions. The hardware takes care of merging the partial register (for instance any of AL, AH or AX is merged into the RAX destination register). This eliminates the special allocation window used to insert merge micro-operation.

From Skylake to Ice Lake microarchitectures, operations that access *H registers (i.e., AH, BH, CH, DH) are executed exclusively on ports 1 and 5.

The *H micro-ops are executed with one cycle latency; however, one cycle of *additional* delay is required for ensuing UOPs because they depend on the results of the *H operation. This additional delay is required due to potential data swapping. A swap might happen, for example, with the instruction "Add AH, BL", or "ADD AL, BH." The pipeline functionality is illustrated in Figure 2-4.

Beginning with the Golden Cove Microarchitecture, the *H operations are limited to Port 1 (port1) with three cycles of latency. This penalty on *H operations helped performance improvement and timing requirements of the Golden Cove microarchitecture.

For more information about Golden Cove microarchitecture, see Section 2.4.1. Figure 2-2 shows the flow.

A closer look at the INT execution ports in Figure 3-1 shows the *H operation limited to Port 1:

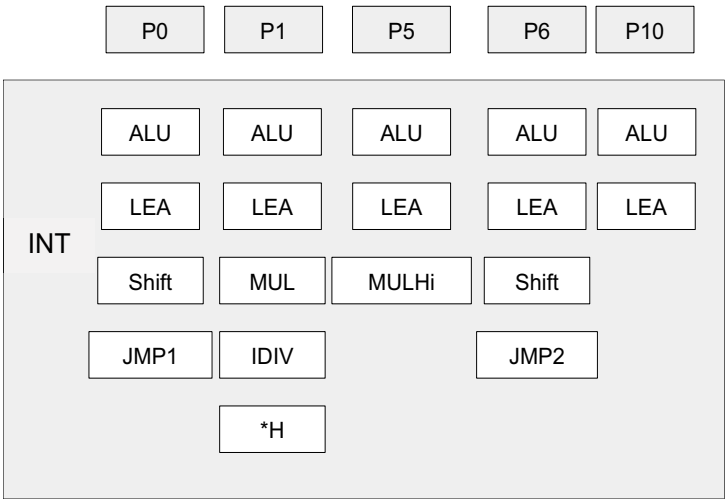


Figure 3-1. INT Execution Ports Within the Processor Core Pipeline

3.5.2.4 Partial XMM Register Stalls

Partial register stalls can also apply to XMM registers. The following SSE and SSE2 instructions update only part of the destination register:

MOVL/HPD XMM, MEM64
MOVL/HPS XMM, MEM32
MOVSS/SD between registers

Using these instructions creates a dependency chain between the unmodified part of the register and the modified part of the register. This dependency chain can cause performance loss.

Example 3-24 illustrates the use of MOVZX to avoid a partial register stall when packing three byte values into a register.

Follow these recommendations to avoid stalls from partial updates to XMM registers:

- Avoid using instructions which update only part of the XMM register.
- If a 64-bit load is needed, use the **MOVSD** or **MOVQ** instruction.
- If 2 64-bit loads are required to the same register from non continuous locations, use **MOVSD/MOVHPD** instead of **MOVLPD/MOVHPD**.
- When copying the XMM register, use the following instructions for full register copy, even if you only want to copy some of the source register data:

  MOVAPS
  MOVAPD
  MOVDQA

Example 3-24. Avoiding Partial Register Stalls in SIMD Code

Using movlpd for memory transactions and movsd between register copies Causing Partial Register Stall	Using movsd for memory and movapd between register copies Avoid Delay

mov edx, x
mov ecx, count
movlpdxmm3,_1_
movlpd xmm2,_1pt5_
align 16
lp:
movlpd xmm0, [edx]
addsd xmm0, xmm3
movsd xmm1, xmm2
subsd xmm1, [edx]
mulsd xmm0, xmm1
movsd [edx], xmm0
add edx, 8
dec ecx
jnz lp
	mov edx, x
	mov ecx, count
	movsd xmm3,_1_
	movsd xmm2, _1pt5_
	align 16
	lp:
	movsd xmm0, [edx]
	addsd xmm0, xmm3
	movapd xmm1, xmm2
	subsd xmm1, [edx]
	mulsd xmm0, xmm1
	movsd [edx], xmm0
	add edx, 8
	dec ecx
	jnz lp

3.5.2.5 Partial Flag Register Stalls

A "partial flag register stall" occurs when an instruction modifies a part of the flag register and the following instruction is dependent on the outcome of the flags. This happens most often with shift instructions (**SAR, SAL, SHR, SHL**). The flags are not modified in the case of a zero shift count, but the shift count is usually known only at execution time. The front end stalls until the instruction is retired.

Other instructions that can modify some part of the flag register include CMPXCHG8B, various rotate instructions, STC, and STD. An example of assembly with a partial flag register stall and alternative code without the stall is shown in Example 3-25.

In processors based on Intel Core microarchitecture, shift immediate by 1 is handled by special hardware so it does not experience partial flag stall.

Example 3-25. Avoiding Partial Flag Register Stalls

Partial Flag Register Stall	Avoiding Partial Flag Register Stall
xor eax, eax mov ecx, a sar ecx, 2 setz al ;SAR can update carry causing a stall	or eax, eax mov ecx, a sar ecx, 2 test ecx, ecx ; test always updates all flags setz al ;No partial reg or flag stall,

In Sandy Bridge microarchitecture, the cost of partial flag access is replaced by the insertion of a micro-op instead of a stall. However, it is still recommended to use less of instructions that write only to some of the flags (such as INC, DEC, SET CL) before instructions that can write flags conditionally (such as SHIFT CL).

Example 3-26 compares two techniques to implement the addition of very large integers (e.g., 1024 bits). The alternative sequence on the right side of Example 3-26 will be faster than the left side on Sandy Bridge microarchitecture, but it will experience partial flag stalls on prior microarchitectures.

Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture

Save partial flag register to avoid stall	Simplified code sequence

lea rsi, [A]
lea rdi, [B]
xor rax, rax
mov rcx, 16 ; 16*64 =1024 bit
lp_64bit:
add rax, [rsi]
adc rax, [rdi]
mov [rdi], rax
setc al ;save carry for next iteration
movzx rax, al
add rsi, 8
add rdi, 8
dec rcx
jnz lp_64bit
	lea rsi, [A]
	lea rdi, [B]
	xor rax, rax
	mov rcx, 16
	lp_64bit:
	add rax, [rsi]
	adc rax, [rdi]
	mov [rdi], rax
	lea rsi, [rsi+8]
	lea rdi, [rdi+8]
	dec rcx
	jnz lp_64bit

3.5.2.6 Floating-Point/SIMD Operands

Moves that write a portion of a register can introduce unwanted dependences. The **MOVSD REG, REG** instruction writes only the bottom 64 bits of a register, not all 128 bits. This introduces a dependence on the preceding instruction that produces the upper 64 bits (even if those bits are not longer wanted). The dependence inhibits register renaming, and thereby reduces parallelism.

Use **MOVAPD** as an alternative; it writes all 128 bits. Even though this instruction has a longer latency, the μops for **MOVAPD** use a different execution port and this port is more likely to be free. The change can impact performance. There may be exceptional cases where the latency matters more than the dependence or the execution port.

Assembly/Compiler Coding Rule 39. (M impact, ML generality) *Avoid introducing dependences with partial floating-point register writes, e.g. from the* **MOVSD XMMREG1, XMMREG2** *instruction. Use the* **MOVAPD XMMREG1, XMMREG2** *instruction instead.*

The **MOVSD XMMREG, MEM** instruction writes all 128 bits and breaks a dependence.

3.5.3 VECTORIZATION

This section provides a brief summary of optimization issues related to vectorization. There is more detail in the chapters that follow.

Vectorization is a program transformation that allows special hardware to perform the same operation on multiple data elements at the same time. Successive processor generations have provided vector support through the MMX technology, Intel Streaming SIMD Extensions (Intel SSE), Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3) and Intel Supplemental Streaming SIMD Extensions 3 (Intel SSSE3).

Vectorization is a special case of SIMD, a term defined in Flynn's architecture taxonomy to denote a single instruction stream capable of operating on multiple data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating-point data elements in Intel SSE and two double-precision floating-point data elements in Intel SSE2 to sixteen byte operations in a 128-bit register in Intel SSE2. Thus, vector length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

The Intel C++ Compiler supports vectorization in three ways:

- The compiler may be able to generate SIMD code without intervention from the user.
- The can user insert pragmas to help the compiler realize that it can vectorize the code.
- The user can write SIMD code explicitly using intrinsics and C++ classes.

To help enable the compiler to generate SIMD code, avoid global pointers and global variables. These issues may be less troublesome if all modules are compiled simultaneously, and whole-program optimization is used.

User/Source Coding Rule 2. (H impact, M generality) *Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.*

User/Source Coding Rule 3. (M impact, ML generality) *Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.*

The integer part of the SIMD instruction set extensions cover 8-bit,16-bit and 32-bit operands. Not all SIMD operations are supported for 32 bits, meaning that some source code will not be able to be vectorized at all unless smaller operands are used.

User/Source Coding Rule 4. (M impact, ML generality) *Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches.*

User/Source Coding Rule 5. (M impact, ML generality) *Keep induction (loop) variable expressions simple.*

3.5.4 OPTIMIZATION OF PARTIALLY VECTORIZABLE CODE

Frequently, a program contains a mixture of vectorizable code and some routines that are non-vectorizable. A common situation of partially vectorizable code involves a loop structure which include mixtures of vectorized code and unvectorizable code. This situation is depicted in Figure 3-2.

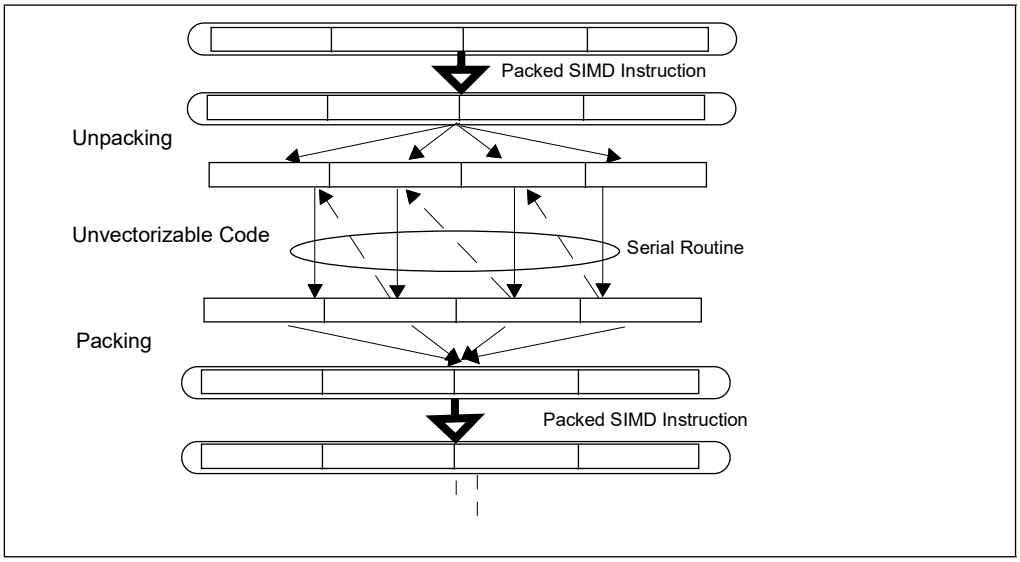


Figure 3-2. Generic Program Flow of Partially Vectorized Code

It generally consists of five stages within the loop:

- Prolog.
- Unpacking vectorized data structure into individual elements.
- Calling a unvectorizable routine to process each element serially.
- Packing individual result into vectorized data structure.
- Epilogue.

This section discusses techniques that can reduce the cost and bottleneck associated with the packing/unpacking stages in these partially vectorize code.

Example 3-27 shows a reference code template that is representative of partially vectorizable coding situations that also experience performance issues. The unvectorizable portion of code is represented generically by a sequence of calling a serial function named “foo” multiple times. This generic example is referred to as “shuffle with store forwarding”, because the problem generally involves an unpacking stage that shuffles data elements between register and memory, followed by a packing stage that can experience store forwarding issue.

There are more than one useful techniques that can reduce the store-forwarding bottleneck between the serialized portion and the packing stage. The following sub-sections presents alternate techniques to deal with the packing, unpacking, and parameter passing to serialized function calls.

Example 3-27. Reference Code Template for Partially Vectorizable Program

```
// Prolog  ///////////////////////////////
push ebp
mov ebp, esp

// Unpacking  /////////////////////////////
sub ebp, 32
and ebp, 0xfffffff0
movaps [ebp], xmm0
```

Example 3-27. Reference Code Template for Partially Vectorizable Program (Contd.)

```
// Serial operations on components ///////
sub ebp, 4
mov eax, [ebp+4]
mov [ebp], eax
call foo
mov [ebp+16+4], eax
mov eax, [ebp+8]
mov [ebp], eax
call foo
mov [ebp+16+4+4], eax

mov eax, [ebp+12]
mov [ebp], eax
call foo
mov [ebp+16+8+4], eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
mov [ebp+16+12+4], eax

// Packing ////////////////////////////////
movaps xmm0, [ebp+16+4]
// Epilog ////////////////////////////////
pop ebp
ret
```

3.5.4.1 Alternate Packing Techniques

The packing method implemented in the reference code of Example 3-27 will experience delay as it assembles 4 doubleword result from memory into an XMM register due to store-forwarding restrictions.

Three alternate techniques for packing, using different SIMD instruction to assemble contents in XMM registers are shown in Example 3-28. All three techniques avoid store-forwarding delay by satisfying the restrictions on data sizes between a preceding store and subsequent load operations.

Example 3-28. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty

Packing Method 1	Packing Method 2	Packing Method 3

movd xmm0, [ebp+16+4]
movd xmm1, [ebp+16+8]
movd xmm2, [ebp+16+12]
movd xmm3, [ebp+12+16+4]
punpckldq xmm0, xmm1
punpckldq xmm2, xmm3
punpckldq xmm0, xmm2
	movd xmm0, [ebp+16+4]
	movd xmm1, [ebp+16+8]
	movd xmm2, [ebp+16+12]
	movd xmm3, [ebp+12+16+4]
	psllq xmm3, 32
	orps xmm2, xmm3
	psllq xmm1, 32
	orps xmm0, xmm1movlhps xmm0, xmm2
		movd xmm0, [ebp+16+4]
		movd xmm1, [ebp+16+8]
		movd xmm2, [ebp+16+12]
		movd xmm3, [ebp+12+16+4]
		movlhps xmm1,xmm3
		psllq xmm1, 32
		movlhps xmm0, xmm2
		orps xmm0, xmm1

3.5.4.2 Simplifying Result Passing

In Example 3-27, individual results were passed to the packing stage by storing to contiguous memory locations. Instead of using memory spills to pass four results, result passing may be accomplished by using either one or more

registers. Using registers to simplify result passing and reduce memory spills can improve performance by varying degrees depending on the register pressure at runtime.

Example 3-29 shows the coding sequence that uses four extra XMM registers to reduce all memory spills of passing results back to the parent routine. However, software must observe the following conditions when using this technique:

- There is no register shortage.
- If the loop does not have many stores or loads but has many computations, this technique does not help performance. This technique adds work to the computational units, while the store and loads ports are idle.

Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing

```
mov eax, [ebp+4]
mov [ebp], eax
call foo
movd xmm0, eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
movd xmm1, eax


mov eax, [ebp+12]
mov [ebp], eax
call foo
movd xmm2, eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
movd xmm3, eax
```

3.5.4.3 Stack Optimization

In Example 3-27, an input parameter was copied in turn onto the stack and passed to the unvectorizable routine for processing. The parameter passing from consecutive memory locations can be simplified by a technique shown in Example 3-30.

Example 3-30. Stack Optimization Technique to Simplify Parameter Passing

```
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax
```

Example 3-30. Stack Optimization Technique to Simplify Parameter Passing (Contd.)

```
add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
```

Stack Optimization can only be used when:

- The serial operations are function calls. The function “foo” is declared as: **INT FOO(INT A)**. The parameter is passed on the stack.
- The order of operation on the components is from last to first.

Note the call to FOO and the advance of EDP when passing the vector elements to FOO one by one from last to first.

3.5.4.4 Tuning Considerations

Tuning considerations for situations represented by looping of Example 3-27 include:

- Applying one of more of the following combinations:
  - Choose an alternate packing technique.
  - Consider a technique to simply result-passing.
  - Consider the stack optimization technique to simplify parameter passing.
- Minimizing the average number of cycles to execute one iteration of the loop.
- Minimizing the per-iteration cost of the unpacking and packing operations.

The speed improvement by using the techniques discussed in this section will vary, depending on the choice of combinations implemented and characteristics of the non-vectorizable routine. For example, if the routine “foo” is short (representative of tight, short loops), the per-iteration cost of unpacking/packing tend to be smaller than situations where the non-vectorizable code contain longer operation or many dependencies. This is because many iterations of short, tight loop can be in flight in the execution core, so the per-iteration cost of packing and unpacking is only partially exposed and appear to cause very little performance degradation.

Evaluation of the per-iteration cost of packing/unpacking should be carried out in a methodical manner over a selected number of test cases, where each case may implement some combination of the techniques discussed in this section. The per-iteration cost can be estimated by:

- Evaluating the average cycles to execute one iteration of the test case.
- Evaluating the average cycles to execute one iteration of a base line loop sequence of non-vectorizable code.

Example 3-31 shows the base line code sequence that can be used to estimate the average cost of a loop that executes non-vectorizable routines.

Example 3-31. Base Line Code Sequence to Estimate Loop Overhead

```
push ebp
mov ebp, esp
sub ebp, 4

mov [ebp], edi
call foo
```

Example 3-31. Base Line Code Sequence to Estimate Loop Overhead (Contd.)

```
mov [ebp], edi
call foo

mov [ebp], edi
call foo


mov [ebp], edi
call foo

add ebp, 4
pop ebp
ret
```

The average per-iteration cost of packing/unpacking can be derived from measuring the execution times of a large number of iterations by:

((Cycles to run TestCase) - (Cycles to run equivalent baseline sequence)) / (Iteration count).

For example, using a simple function that returns an input parameter (representative of tight, short loops), the per-iteration cost of packing/unpacking may range from slightly more than 7 cycles (the shuffle with store forwarding case, Example 3-27) to ~0.9 cycles (accomplished by several test cases). Across 27 test cases (consisting of one of the alternate packing methods, no result-simplification/simplification of either 1 or 4 results, no stack optimization or with stack optimization), the average per-iteration cost of packing/unpacking is about 1.7 cycles.

Generally speaking, packing method 2 and 3 (see Example 3-28) tend to be more robust than packing method 1; the optimal choice of simplifying 1 or 4 results will be affected by register pressure of the runtime and other relevant microarchitectural conditions.

Note that the numeric discussion of per-iteration cost of packing/packing is illustrative only. It will vary with test cases using a different base line code sequence and will generally increase if the non-vectorizable routine requires longer time to execute because the number of loop iterations that can reside in flight in the execution core decreases.

3.6 OPTIMIZING MEMORY ACCESSES

This section discusses guidelines for optimizing code and data memory accesses. The most important recommendations are:

- Execute load and store operations within available execution bandwidth.
- Enable forward progress of speculative execution.
- Enable store forwarding to proceed.
- Align data, paying attention to data layout and stack alignment.
- Place code and data on separate pages.
- Enhance data locality.
- Use prefetching and cacheability control instructions.
- Enhance code locality and align branch targets.
- Take advantage of write combining.

3.6.1 LOAD AND STORE EXECUTION BANDWIDTH

Typically, loads and stores are the most frequent operations in a workload, up to 40% of the instructions in a workload carrying load or store intent are not uncommon. Each generation of microarchitecture provides multiple buffers to support executing load and store operations while there are instructions in flight. These buffers were comprised of 128-bit wide entries for the Sandy Bridge and Ivy Bridge microarchitectures. The size was increased to 256-bit in Haswell, Broadwell and Skylake Client microarchitectures; and to 512-bit in Skylake Server, Cascade Lake, Cascade Lake Advanced Performance, and Ice Lake Client microarchitectures. To maximize performance, it is best to use the largest width available in the platform.

3.6.1.1 Making Use of Load Bandwidth in Sandy Bridge Microarchitecture

While prior microarchitecture has one load port (port 2), Sandy Bridge microarchitecture can load from port 2 and port 3. Thus two load operations can be performed every cycle and doubling the load throughput of the code. This improves code that reads a lot of data and does not need to write out results to memory very often (Port 3 also handles store-address operation). To exploit this bandwidth, the data has to stay in the L1 data cache or it should be accessed sequentially, enabling the hardware prefetchers to bring the data to the L1 data cache in time.

Consider the following C code example of adding all the elements of an array:

```
int buff[BUFF_SIZE];
int sum = 0;

for (i=0;i<BUFF_SIZE;i++){
    sum+=buff[i];
}
```

Alternative 1 is the assembly code generated by the Intel compiler for this C code, using the optimization flag for Nehalem microarchitecture. The compiler vectorizes execution using Intel SSE instructions. In this code, each ADD operation uses the result of the previous ADD operation. This limits the throughput to one load and ADD operation per cycle. Alternative 2 is optimized for Sandy Bridge microarchitecture by enabling it to use the additional load bandwidth. The code removes the dependency among ADD operations, by using two registers to sum the array values. Two load and two ADD operations can be executed every cycle.

Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution
xor eax, eax pxor xmm0, xmm0 lea rsi, buff	xor eax, eax pxor xmm0, xmm0 pxor xmm1, xmm1 lea rsi, buff

Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture (Contd.)

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution

loop_start:
paddd xmm0, [rsi+4*rax]
paddd xmm0, [rsi+4*rax+16]
paddd xmm0, [rsi+4*rax+32]
paddd xmm0, [rsi+4*rax+48]
paddd xmm0, [rsi+4*rax+64]
paddd xmm0, [rsi+4*rax+80]
paddd xmm0, [rsi+4*rax+96]
paddd xmm0, [rsi+4*rax+112]
add eax, 32
cmp eax, BUFF_SIZE
jl loop_start
sum_partials:
movdqa xmm1, xmm0
psrldq xmm1, 8
paddd xmm0, xmm1
movdqa xmm2, xmm0
psrldq xmm2, 4
paddd xmm0, xmm2
movd [sum], xmm0
	loop_start:
	paddd xmm0, [rsi+4*rax]
	paddd xmm1, [rsi+4*rax+16]
	paddd xmm0, [rsi+4*rax+32]
	paddd xmm1, [rsi+4*rax+48]
	paddd xmm0, [rsi+4*rax+64]
	paddd xmm1, [rsi+4*rax+80]
	paddd xmm0, [rsi+4*rax+96]
	paddd xmm1, [rsi+4*rax+112]
	add eax, 32
	cmp eax, BUFF_SIZE
	jl loop_start
	sum_partials:
	paddd xmm0, xmm1
	movdqa xmm1, xmm0
	psrldq xmm1, 8
	paddd xmm0, xmm1
	movdqa xmm2, xmm0
	psrldq xmm2, 4
	paddd xmm0, xmm2
	movd [sum], xmm0

3.6.1.2 L1D Cache Latency in Sandy Bridge Microarchitecture

Load latency from L1D cache may vary. The best case if 4 cycles, which apply to load operations to general purpose registers using one of the following:

- One register.
- A base register plus an offset that is smaller than 2048.

Consider the pointer-chasing code example in Example 3-33.

Example 3-33. Index versus Pointers in Pointer-Chasing Code

Traversing through indexes	Traversing through pointers

// C code example
index = buffer.m_buff[index].next_index;
// ASM example
loop:
shl rbx, 6
mov rbx, 0x20(rbx+rcx)
dec rax
cmp rax, -1
jne loop
	// C code example
	node = node->pNext;
	// ASM example
	loop:
	mov rdx, [rdx]
	dec rax
	cmp rax, -1
	jne loop

The left side implements pointer chasing via traversing an index. Compiler then generates the code shown below addressing memory using base+index with an offset. The right side shows compiler generated code from pointer de-referencing code and uses only a base register.

The code on the right side is faster than the left side across Sandy Bridge microarchitecture and prior microarchitecture. However the code that traverses index will be slower on Sandy Bridge microarchitecture relative to prior microarchitecture.

3.6.1.3 Handling L1D Cache Bank Conflict

In the Sandy Bridge microarchitecture, the internal organization of the L1D cache may manifest a situation when two load micro-ops whose addresses have a bank conflict. When a bank conflict is present between two load operations, the more recent one will be delayed until the conflict is resolved. A bank conflict happens when two simultaneous load operations have the same bit 2-5 of their linear address but they are not from the same set in the cache (bits 6 - 12).

Bank conflicts should be handled only if the code is bound by load bandwidth. Some do not cause any performance degradation since they are hidden by other performance limiters. Eliminating such bank conflicts does not improve performance.

The L1D cache bank conflict issue does not apply to Haswell microarchitecture.

The following example demonstrates bank conflict and how to modify the code and avoid them. It uses two source arrays with a size that is a multiple of cache line size. When loading an element from A and the counterpart element from B the elements have the same offset in their cache lines; therefore, a bank conflict may happen.

Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy

```
int A[128];
int B[128];
int C[128];
for (i=0;i<128;i+=4){
    C[i]=A[i]+B[i];      the loads from A[i] and B[i] collide
    C[i+1]=A[i+1]+B[i+1];
    C[i+2]=A[i+2]+B[i+2];
    C[i+3]=A[i+3]+B[i+3];
}
```

```
// Code with Bank Conflicts
    xor rcx, rcx
    lea r11, A
    lea r12, B
    lea r13, C
loop:
    lea esi, [rcx*4]
    movsxd rsi, esi
    mov edi, [r11+rsi*4]
    add edi, [r12+rsi*4]
    mov r8d, [r11+rsi*4+4]
    add r8d, [r12+rsi*4+4]
    mov r9d, [r11+rsi*4+8]
    add r9d, [r12+rsi*4+8]
    mov r10d, [r11+rsi*4+12]
    add r10d, [r12+rsi*4+12]
```

```
// Code without Bank Conflicts
    xor rcx, rcx
    lea r11, A
    lea r12, B
    lea r13, C
loop:
    lea esi, [rcx*4]
    movsxd rsi, esi
    mov edi, [r11+rsi*4]
    mov r8d, [r11+rsi*4+4]
    add edi, [r12+rsi*4]
    add r8d, [r12+rsi*4+4]
    mov r9d, [r11+rsi*4+8]
    mov r10d, [r11+rsi*4+12]
    add r9d, [r12+rsi*4+8]
    add r10d, [r12+rsi*4+12]
```

Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy (Contd.)

mov [r13+rsi*4], edi
inc ecx
mov [r13+rsi*4+4], r8d
mov [r13+rsi*4+8], r9d
mov [r13+rsi*4+12], r10d
cmp ecx, LEN
jb loop
	inc ecx
	mov [r13+rsi*4], edi
	mov [r13+rsi*4+4], r8d
	mov [r13+rsi*4+8], r9d
	mov [r13+rsi*4+12], r10d
	cmp ecx, LEN
	jb loop

Bank conflicts may occur with the introduction of the third load port in the Golden Cove microarchitecture. In this microarchitecture, conflicts happen between three loads with the same bits 2-5 of their linear address even if they access the same set of the cache. Up to two loads can access the same cache bank without a conflict; however, a third load accessing the same bank must be delayed. The bank conflicts do not apply to 512-bit wide loads because their bandwidth is limited to two per cycle.

Recommendation: In the Golden Cove microarchitecture, bank conflicts often happen when multiple loads access the same memory location. Whenever possible, avoid reading the same memory location within a tight loop or using multiple load operations. Commonly used memory locations are better kept in the registers to prevent potential bank conflict penalty.

3.6.2 MINIMIZE REGISTER SPILLS

When a piece of code has more live variables than the processor can keep in general purpose registers, a common method is to hold some of the variables in memory. This method is called register spill. The effect of L1D cache latency can negatively affect the performance of this code. The effect can be more pronounced if the address of register spills uses the slower addressing modes.

One option is to spill general purpose registers to XMM registers. This method is likely to improve performance also on previous processor generations. The following example shows how to spill a register to an XMM register rather than to memory.

Example 3-35. Using XMM Register in Lieu of Memory for Register Spills

Register spills into memory	Register spills into XMM

loop:
mov rdx, [rsp+0x18]
movdqa xmm0, [rdx]
movdqa xmm1, [rsp+0x20]
pcmpeqd xmm1, xmm0
pmovmskb eax, xmm1
test eax, eax
jne end_loop
movzx rcx, [rbx+0x60]
	movq xmm4, [rsp+0x18]
	mov rcx, 0x10
	movq xmm5, rcx
	loop:
	movq rdx, xmm4
	movdqa xmm0, [rdx]
	movdqa xmm1, [rsp+0x20]
	pcmpeqd xmm1, xmm0
	pmovmskb eax, xmm1
	test eax, eax
	jne end_loop
	movzx rcx, [rbx+0x60]

Example 3-35. Using XMM Register in Lieu of Memory for Register Spills (Contd.)

Register spills into memory	Register spills into XMM

add qword ptr[rsp+0x18], 0x10
add rdi, 0x4
movzx rdx, di
sub rcx, 0x4
add rsi, 0x1d0
cmp rdx, rcx
jle loop
	padd xmm4, xmm5
	add rdi, 0x4
	movzx rdx, di
	sub rcx, 0x4
	add rsi, 0x1d0
	cmp rdx, rcx
	jle loop

3.6.3 ENHANCE SPECULATIVE EXECUTION AND MEMORY DISAMBIGUATION

Prior to Intel Core microarchitecture, when code contains both stores and loads, the loads cannot be issued before the address of the older stores is known. This rule ensures correct handling of load dependencies on preceding stores.

The Intel Core microarchitecture contains a mechanism that allows some loads to be executed speculatively in the presence of older unknown stores. The processor later checks if the load address overlapped with an older store whose address was unknown at the time the load executed. If the addresses do overlap, then the processor re-executes the load and all succeeding instructions.

Example 3-36 illustrates a situation that the compiler cannot be sure that "Ptr->Array" does not change during the loop. Therefore, the compiler cannot keep "Ptr->Array" in a register as an invariant and must read it again in every iteration. Although this situation can be fixed in software by a rewriting the code to require the address of the pointer is invariant, memory disambiguation improves performance without rewriting the code.

Example 3-36. Loads Blocked by Stores of Unknown Address

C Code	Assembly Sequence

struct AA {
AA ** array;
};
void nullify_array (AA *Ptr, DWORD Index, AA *ThisPtr)
{
while (Ptr->Array[--Index] != ThisPtr)
{
Ptr->Array[Index] = NULL ;
} ;
} ;
	nullify_loop:
	mov dword ptr [eax], 0
	mov edx, dword ptr [edi]
	sub ecx, 4
	cmp dword ptr [ecx+edx], esi
	lea eax, [ecx+edx]
	jne nullify_loop

It is possible to disable speculative store bypass with the IA32_SPEC_CTRL.SSBD MSR.

Additional information on this topic can be found on the Software Security Guidance page.

3.6.4 STORE FORWARDING

The processor's memory system only sends stores to memory (including cache) after store retirement. However, store data can be forwarded from a store to a subsequent load from the same address to give a much shorter store-load latency.

There are two kinds of requirements for store forwarding. If these requirements are violated, store forwarding cannot occur and the load must get its data from the cache (so the store must write its data back to the cache first). This incurs a penalty that is largely related to pipeline depth of the underlying micro-architecture.

The first requirement pertains to the size and alignment of the store-forwarding data. This restriction is likely to have high impact on overall application performance. Typically, a performance penalty due to violating this restriction can be prevented. The store-to-load forwarding restrictions vary from one microarchitecture to another. Several examples of coding pitfalls that cause store-forwarding stalls and solutions to these pitfalls are discussed in detail in Section 3.6.4.1 The second requirement is the availability of data, discussed in Section 3.6.4.2 A good practice is to eliminate redundant load operations.

It may be possible to keep a temporary scalar variable in a register and never write it to memory. Generally, such a variable must not be accessible using indirect pointers. Moving a variable to a register eliminates all loads and stores of that variable and eliminates potential problems associated with store forwarding. However, it also increases register pressure.

Load instructions tend to start chains of computation. Since the out-of-order engine is based on data dependence, load instructions play a significant role in the engine's ability to execute at a high rate. Eliminating loads should be given a high priority.

If a variable does not change between the time when it is stored and the time when it is used again, the register that was stored can be copied or used directly. If register pressure is too high, or an unseen function is called before the store and the second load, it may not be possible to eliminate the second load.

Assembly/Compiler Coding Rule 40. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible. Passing arguments on the stack requires a store followed by a reload. While this sequence is optimized in hardware by providing the value to the load directly from the memory order buffer without the need to access the data cache if permitted by store-forwarding restrictions, floating-point values incur a significant latency in forwarding. Passing floating-point arguments in (preferably XMM) registers should save this long latency operation.*

Parameter passing conventions may limit the choice of which parameters are passed in registers which are passed on the stack. However, these limitations may be overcome if the compiler has control of the compilation of the whole binary (using whole-program optimization).

3.6.4.1 Store-to-Load-Forwarding Restriction on Size and Alignment

Data size and alignment restrictions for store-forwarding apply to processors based on Intel Core microarchitecture, Intel Core 2 Duo, Intel Core Solo and Pentium M processors. The performance penalty for violating store-forwarding restrictions is less for shorter-pipelined machines.

Store-forwarding restrictions vary with each microarchitecture. The following rules help satisfy size and alignment restrictions for store forwarding:

Assembly/Compiler Coding Rule 41. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data.*

Assembly/Compiler Coding Rule 42. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data.*

A load that forwards from a store must wait for the store's data to be written to the store buffer before proceeding, but other, unrelated loads need not wait.

Assembly/Compiler Coding Rule 43. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. This is better than incurring the penalties of a failed store-forward.*

Assembly/Compiler Coding Rule 44. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed.*

Example 3-37 depicts several store-forwarding situations in which small loads follow large stores. The first three load operations illustrate the situations described in Rule 44. However, the last load operation gets data from store-forwarding without problem.

Example 3-37. Situations Showing Small Loads After Large Store

```
mov [EBP],'abcd'
mov AL, [EBP]       ; Not blocked - same alignment
mov BL, [EBP + 1]   ; Blocked
mov CL, [EBP + 2]   ; Blocked
mov DL, [EBP + 3]   ; Blocked
mov AL, [EBP]       ; Not blocked - same alignment
                    ; n.b. passes older blocked loads
```

Example 3-38 illustrates a store-forwarding situation in which a large load follows several small stores. The data needed by the load operation cannot be forwarded because all of the data that needs to be forwarded is not contained in the store buffer. Avoid large loads after small stores to the same area of memory.

Example 3-38. Non-forwarding Example of Large Load After Small Store

```
mov [EBP], 'a'
mov [EBP + 1], 'b'
mov [EBP + 2], 'c'
mov [EBP + 3], 'd'
mov EAX, [EBP]  ; Blocked
    ; The first 4 small store can be consolidated into
    ; a single DWORD store to prevent this non-forwarding
    ; situation.
```

Example 3-39 illustrates a stalled store-forwarding situation that may appear in compiler generated code. Sometimes a compiler generates code similar to that shown in Example 3-39 to handle a spilled byte to the stack and convert the byte to an integer value.

Example 3-39. A Non-forwarding Situation in Compiler Generated Code

```
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h]  ; Stall
and eax, 0xff                 ; Converting back to byte value
```

Example 3-40 offers two alternatives to avoid the non-forwarding situation shown in Example 3-39.

Example 3-40. Two Ways to Avoid Non-forwarding Situation in Example 3-39

```
; A. Use MOVZ instruction to avoid large load after small
; store, when spills are ignored.
movz eax, bl                    ; Replaces the last three instructions
; B. Use MOVZ instruction and handle spills to the stack
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h]   ; Not blocked
```

When moving data that is smaller than 64 bits between memory locations, 64-bit or 128-bit SIMD register moves are more efficient (if aligned) and can be used to avoid unaligned loads. Although floating-point registers allow the movement of 64 bits at a time, floating-point instructions should not be used for this purpose, as data may be inadvertently modified.

As an additional example, consider the cases in Example 3-41.

Example 3-41. Large and Small Load Stalls

```
; A. Large load stall
mov     mem, eax      ; Store dword to address “MEM"
mov     mem + 4, ebx  ; Store dword to address “MEM + 4"
fld     mem           ; Load qword at address “MEM", stalls
; B. Small Load stall
fstp    mem           ; Store qword to address “MEM"
mov     bx, mem+2     ; Load word at address “MEM + 2", stalls
mov     cx, mem+4     ; Load word at address “MEM + 4", stalls
```

In the first case (A), there is a large load after a series of small stores to the same area of memory (beginning at memory address **MEM**). The large load will stall.

The **FLD** must wait for the stores to write to memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

In the second case (B), there is a series of small loads after a large store to the same area of memory (beginning at memory address **MEM**). The small loads will stall.

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible.

Store forwarding restrictions for processors based on Intel Core microarchitecture is listed in Table 3-4.

Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	16	word aligned	8, 16	not stalled

Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	16	not word aligned	8	stalled
To Natural size	32	dword aligned	8, 32	not stalled
To Natural size	32	not dword aligned	8	stalled
To Natural size	32	word aligned	16	not stalled
To Natural size	32	not word aligned	16	stalled
To Natural size	64	qword aligned	8, 16, 64	not stalled
To Natural size	64	not qword aligned	8, 16	stalled
To Natural size	64	dword aligned	32	not stalled
To Natural size	64	not dword aligned	32	stalled
To Natural size	128	dqword aligned	8, 16, 128	not stalled
To Natural size	128	not dqword aligned	8, 16	stalled
To Natural size	128	dword aligned	32	not stalled
To Natural size	128	not dword aligned	32	stalled
To Natural size	128	qword aligned	64	not stalled
To Natural size	128	not qword aligned	64	stalled
Unaligned, start byte 1	32	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	32	not byte 0 of store	8, 16	stalled
Unaligned, start byte 1	64	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	64	not byte 0 of store	8, 16, 32	stalled
Unaligned, start byte 1	64	byte 0 of store	64	stalled
Unaligned, start byte 7	32	byte 0 of store	8	not stalled
Unaligned, start byte 7	32	not byte 0 of store	8	not stalled
Unaligned, start byte 7	32	don’t care	16, 32	stalled
Unaligned, start byte 7	64	don’t care	16, 32, 64	stalled

3.6.4.2 Store-Forwarding Restriction on Data Availability

The value to be stored must be available before the load operation can be completed. If this restriction is violated, the execution of the load will be delayed until the data is available. This delay causes some execution resources to be used unnecessarily, and that can lead to sizable but non-deterministic delays. However, the overall impact of this problem is much smaller than that from violating size and alignment requirements.

In modern microarchitectures, hardware predicts when loads are dependent on and get their data forwarded from preceding stores. These predictions can significantly improve performance. However, if a load is scheduled too soon after the store it depends on or if the generation of the data to be stored is delayed, there can be a significant penalty.

There are several cases in which data is passed through memory, and the store may need to be separated from the load:

- Spills, save and restore registers in a stack frame.
- Parameter passing.
- Global and volatile variables.
- Type conversion between integer and floating-point.
- When compilers do not analyze code that is inlined, forcing variables that are involved in the interface with inlined code to be in memory, creating more memory variables and preventing the elimination of redundant loads.

Assembly/Compiler Coding Rule 45. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction - for example, MUL or DIV. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain.*

Example 3-42 shows an example of a loop-carried dependence chain.

Example 3-42. Loop-Carried Dependence Chain

```
for ( i = 0; i < MAX; i++ ) {
    a[i] = b[i] * foo;
    foo = a[i] / 3;
}                       // foo is a loop-carried dependence.
```

Assembly/Compiler Coding Rule 46. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads.*

3.6.5 DATA LAYOUT OPTIMIZATIONS

User/Source Coding Rule 6. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary.*

If the operands are packed in a SIMD instruction, align to the packed element size (64-bit or 128-bit).

Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding. However, compilers might not have this freedom. The C programming language, for example, specifies the order in which structure elements are allocated in memory. For more information, see Section 5.4.

Example 3-43 shows how a data structure could be rearranged to reduce its size.

Example 3-43. Rearranging a Data Structure

```
struct unpacked { /* Fits in 20 bytes due to padding */
    int        a;
    char       b;
    int        c;
    char       d;
    int        e;
};

struct packed {  /* Fits in 16 bytes */
    int        a;
    int        c;
    int        e;
    char       b;
    char       d;
}
```

Cache line size of 64 bytes can impact streaming applications (for example, multimedia). These reference and use data only once before discarding it. Data accesses which sparsely utilize the data within a cache line can result in less efficient utilization of system memory bandwidth. For example, arrays of structures can be decomposed into several arrays to achieve better packing, as shown in Example 3-44.

Example 3-44. Decomposing an Array

```
struct {       /* 1600 bytes */
    int    a, c, e;
    char b, d;
} array_of_struct [100];

struct {       /* 1400 bytes */
    int    a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;


struct {       /* 1200 bytes */
    int   a, c, e;
} hybrid_struct_of_array_ace[100];


struct {       /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

The efficiency of such optimizations depends on usage patterns. If the elements of the structure are all accessed together but the access pattern of the array is random, then **ARRAY_OF_STRUCT** avoids unnecessary prefetch even though it wastes memory.

However, if the access pattern of the array exhibits locality (for example, if the array index is being swept through) then processors with hardware prefetchers will prefetch data from **STRUCT_OF_ARRAY**, even if the elements of the structure are accessed together.

When the elements of the structure are not accessed with equal frequency, such as when element **A** is accessed ten times more often than the other entries, then **STRUCT_OF_ARRAY** not only saves memory, but it also prevents fetching unnecessary data items **B**, **C**, **D**, and **E**.

Using **STRUCT_OF_ARRAY** also enables the use of the SIMD data types by the programmer and the compiler.

Note that **STRUCT_OF_ARRAY** can have the disadvantage of requiring more independent memory stream references. This can require the use of more prefetches and additional address generation calculations. It can also have an impact on DRAM page access efficiency. An alternative, **HYBRID_STRUCT_OF_ARRAY** blends the two approaches. In this case, only 2 separate address streams are generated and referenced: 1 for **HYBRID_STRUCT_OF_ARRAY_ACE** and 1 for **HYBRID_STRUCT_OF_ARRAY_BD**. The second alterative also prevents fetching unnecessary data — assuming that (1) the variables **A**, **C** and **E** are always used together, and (2) the variables **B** and **D** are always used together, but not at the same time as **A**, **C** and **E**.

The hybrid approach ensures:

- Simpler/fewer address generations than **STRUCT_OF_ARRAY.**
- Fewer streams, which reduces DRAM page misses.
- Fewer prefetches due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

Assembly/Compiler Coding Rule 47. (H impact, M generality) *Try to arrange data structures so they permit sequential access.*

If the data is arranged into a set of streams, the automatic hardware prefetcher can prefetch data that will be needed by the application, reducing the effective memory latency. If the data is accessed in a non-sequential manner, the automatic hardware prefetcher cannot prefetch the data. The prefetcher can recognize up to eight concurrent streams. See Chapter 9 for more information on the hardware prefetcher.

User/Source Coding Rule 7. (M impact, L generality) *Beware of false sharing within a cache line (64 bytes).*

3.6.6 STACK ALIGNMENT

Performance penalty of unaligned access to the stack happens when a memory reference splits a cache line. This means that one out of eight spatially consecutive unaligned quadword accesses is always penalized, similarly for one out of 4 consecutive, non-aligned double-quadword accesses, etc.

Aligning the stack may be beneficial any time there are data objects that exceed the default stack alignment of the system. For example, on 32/64bit Linux, and 64bit Windows, the default stack alignment is 16 bytes, while 32bit Windows is 4 bytes.

Assembly/Compiler Coding Rule 48. (H impact, M generality) *Make sure that the stack is aligned at the largest multi-byte granular data type boundary matching the register width.*

Aligning the stack typically requires the use of an additional register to track across a padded area of unknown amount. There is a trade-off between causing unaligned memory references that spanned across a cache line and causing extra general purpose register spills.

The assembly level technique to implement dynamic stack alignment may depend on compilers, and specific OS environment. The reader may wish to study the assembly output from a compiler of interest.

Example 3-45. Examples of Dynamical Stack Alignment

```
// 32-bit environment
    push    ebp ; save ebp
    mov     ebp, esp ; ebp now points to incoming parameters
    andl    esp, $-<N> ;align esp to N byte boundary
    sub     esp, $<stack_size>; reserve space for new stack frame
    .        ; parameters must be referenced off of ebp
    mov     esp, ebp ; restore esp
    pop     ebp ; restore ebp


// 64-bit environment
    sub     esp, $<stack_size +N>
    mov     r13, $<offset_of_aligned_section_in_stack>
    andl    r13, $-<N> ; r13 point to aligned section in stack
    .        ;use r13 as base for aligned data
```

If for some reason it is not possible to align the stack for 64-bits, the routine should access the parameter and save it into a register or known aligned storage, thus incurring the penalty only once.

3.6.7 CAPACITY LIMITS AND ALIASING IN CACHES

There are cases in which addresses with a given stride will compete for some resource in the memory hierarchy.

Typically, caches are implemented to have multiple ways of set associativity, with each way consisting of multiple sets of cache lines (or sectors in some cases). Multiple memory references that compete for the same set of each way in a cache can cause a capacity issue. There are aliasing conditions that apply to specific microarchitectures. Note that first-level cache lines are 64 bytes. Thus, the least significant 6 bits are not considered in alias comparisons.

3.6.8 MIXING CODE AND DATA

The aggressive prefetching and pre-decoding of instructions by Intel processors have two related effects:

- Self-modifying code (SMC) works correctly, according to the Intel architecture processor requirements, but incurs a significant performance penalty. Avoid self-modifying code if possible.
- Placing writable data in the code segment might be impossible to distinguish from self-modifying code. Writable data in the code segment might suffer the same performance penalty as self-modifying code.

Assembly/Compiler Coding Rule 49. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch.*

Tuning Suggestion 1. *In rare cases, a performance problem may be caused by executing data on a code page as instructions. This is very likely to happen when execution is following an indirect branch that is not resident in the trace cache. If this is clearly causing a performance problem, try moving the data elsewhere, or inserting an illegal opcode or a* **PAUSE** *instruction immediately after the indirect branch. Note that the latter two alternatives may degrade performance in some circumstances.*

Assembly/Compiler Coding Rule 50. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate 4-KByte pages or on separate aligned 1-KByte subpages.*

3.6.8.1 Self-Modifying Code (SMC)

Self-modifying code (SMC) that ran correctly on Pentium III processors and prior implementations will run correctly on subsequent implementations. SMC and cross-modifying code (when multiple processors in a multiprocessor system are writing to a code page) should be avoided when high performance is desired.

Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written. In addition, sharing a page containing directly or speculatively executed code with another processor as a data page can trigger an SMC condition causing the entire pipeline of the machine and the trace cache to be cleared.

Dynamic code need not cause the SMC condition if the code written fills up a data page before that page is accessed as code. Dynamically-modified code (for example, from target fix-ups) is likely to suffer from the SMC condition and should be avoided where possible. Avoid the condition by introducing indirect branches and using data tables on data pages (not code pages) using register-indirect calls.

3.6.8.2 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. Example 3-46a shows one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 3-46b shows an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 3-46. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP does not corrupt the RSB
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX


b) Using matched call/ret pair

        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

3.6.9 WRITE COMBINING

Write combining (WC) improves performance in two ways:

- On a write miss to the first-level cache, it allows multiple stores to the same cache line to occur before that cache line is read for ownership (RFO) from further out in the cache/memory hierarchy. Then the rest of line is read, and the bytes that have not been written are combined with the unmodified bytes in the returned line.

- Write combining allows multiple writes to be assembled and written further out in the cache hierarchy as a unit. This saves port and bus traffic. Saving traffic is particularly important for avoiding partial writes to uncached memory.

Processors based on Intel Core microarchitecture have eight write-combining buffers in each core. Beginning with Nehalem microarchitecture, there are 10 buffers available for write-combining. Beginning with Ice Lake Client microarchitecture, there are 12 buffers available for write-combining.

Assembly/Compiler Coding Rule 51. (H impact, L generality) *If an inner loop writes to more than four arrays (four distinct cache lines), apply loop fission to break up the body of the loop so only four arrays are being written to in each iteration of each of the resulting loops.*

Write combining buffers are used for stores of all memory types. They are particularly important for writes to uncached memory: writes to different parts of the same cache line can be grouped into a single, full-cache-line bus transaction instead of going across the bus (since they are not cached) as several partial writes. Avoiding partial writes can have a significant impact on bus bandwidth-bound graphics applications, where graphics buffers are in uncached memory. Separating writes to uncached memory and writes to writeback memory into separate phases can assure that the write combining buffers can fill before getting evicted by other write traffic. Eliminating partial write transactions has been found to have performance impact on the order of 20% for some applications. Because the cache lines are 64 bytes, a write to the bus for 63 bytes will result in partial bus transactions.

When coding functions that execute simultaneously on two threads, reducing the number of writes that are allowed in an inner loop will help take full advantage of write-combining store buffers. For write-combining buffer recommendations for Intel® Hyper-Threading Technology (Intel® HT), see Chapter 11.

Store ordering and visibility are also important issues for write combining. When a write to a write-combining buffer for a previously-unwritten cache line occurs, there will be a read-for-ownership (RFO). If a subsequent write happens to another write-combining buffer, a separate RFO may be caused for that cache line. Subsequent writes to the first cache line and write-combining buffer will be delayed until the second RFO has been serviced to guarantee properly ordered visibility of the writes. If the memory type for the writes is write-combining, there will be no RFO since the line is not cached, and there is no such delay. For details on write-combining, see Chapter 9, “Optimizing Cache Usage”

3.6.10 LOCALITY ENHANCEMENT

Locality enhancement can reduce data traffic originating from an outer-level sub-system in the cache/memory hierarchy. This is to address the fact that the access-cost in terms of cycle-count from an outer level will be more expensive than from an inner level. Typically, the cycle-cost of accessing a given cache level (or memory system) varies across different microarchitectures, processor implementations, and platform components. It may be sufficient to recognize the relative data access cost trend by locality rather than to follow a large table of numeric values of cycle-costs, listed per locality, per processor/platform implementations, etc. The general trend is typically that access cost from an outer sub-system may be approximately 3-10X more expensive than accessing data from the immediate inner level in the cache/memory hierarchy, assuming similar degrees of data access parallelism.

Thus locality enhancement should start with characterizing the dominant data traffic locality. Appendix A, “Application Performance Tools” describes some techniques that can be used to determine the dominant data traffic locality for any workload.

Even if cache miss rates of the last level cache may be low relative to the number of cache references, processors typically spend a sizable portion of their execution time waiting for cache misses to be serviced. Reducing cache misses by enhancing a program’s locality is a key optimization. This can take several forms:

- Blocking to iterate over a portion of an array that will fit in the cache (with the purpose that subsequent references to the data-block [or tile] will be cache hit references).
- Loop interchange to avoid crossing cache lines or page boundaries.

- Loop skewing to make accesses contiguous.

Locality enhancement to the last level cache can be accomplished with sequencing the data access pattern to take advantage of hardware prefetching. This can also take several forms:

- Transformation of a sparsely populated multi-dimensional array into a one-dimension array so memory references occur in a sequential, small-stride pattern that is friendly to the hardware prefetch.
- Optimal tile size and shape selection can further improve temporal data locality by increasing hit rates into the last level cache and reduce memory traffic resulting from the actions of hardware prefetching (see Section 9.5.11).

It is important to avoid operations that work against locality-enhancing techniques. Using the lock prefix heavily can incur large delays when accessing memory, regardless of whether the data is in the cache or in system memory.

User/Source Coding Rule 8. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops.*

Optimizing for one-half of the first-level cache will bring the greatest performance benefit in terms of cycle-cost per data access. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.

3.6.11 NON-TEMPORAL STORE BUS TRAFFIC

Peak system bus bandwidth is shared by several types of bus activities, including reads (from memory), reads for ownership (of a cache line), and writes. The data transfer rate for bus write transactions is higher if 64 bytes are written out to the bus at a time.

Typically, bus writes to Writeback (WB) memory must share the system bus bandwidth with read-for-ownership (RFO) traffic. Non-temporal stores do not require RFO traffic; they do require care in managing the access patterns in order to ensure 64 bytes are evicted at once (rather than evicting several chunks).

Although the data bandwidth of full 64-byte bus writes due to non-temporal stores is twice that of bus writes to WB memory, transferring several chunks wastes bus request bandwidth and delivers significantly lower data bandwidth. This difference is depicted in Examples 3-47 and 3-48.

Example 3-47. Using Non-Temporal Stores and 64-byte Bus Write Transactions

```
#define STRIDESIZE 256
lea ecx, p64byte_Aligned
mov edx, ARRAY_LEN
xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
movntps XMMWORD ptr [ecx + eax+48], xmm0
; 64 bytes is written in one bus transaction
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

Example 3-48. On-temporal Stores and Partial Bus Write Transactions

```
#define STRIDESIZE 256
Lea ecx, p64byte_Aligned
Mov edx, ARRAY_LEN
Xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0


; Storing 48 bytes results in several bus partial transactions
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

3.7 PREFETCHING

Recent Intel processor families employ several prefetching mechanisms to accelerate the movement of data or code and improve performance:

- Hardware instruction prefetcher.
- Software prefetch for data.
- Hardware prefetch for cache lines of data or instructions.

3.7.1 HARDWARE INSTRUCTION FETCHING AND SOFTWARE PREFETCHING

Software prefetching requires a programmer to use PREFETCH hint instructions and anticipate some suitable timing and location of cache misses.

Software PREFETCH operations work the same way as do load from memory operations, with the following exceptions:

- Software PREFETCH instructions retire after virtual to physical address translation is completed.
- If an exception, such as page fault, is required to prefetch the data, then the software prefetch instruction retires without prefetching data.
- Avoid specifying a NULL address for software prefetches.

3.7.2 HARDWARE PREFETCHING FOR FIRST-LEVEL DATA CACHE

Example 3-49 depicts a technique to trigger hardware prefetch. The code demonstrates traversing a linked list and performing some computational work on two members of each element that reside in two different cache lines. Each element is of size 192 bytes. The total size of all elements is larger than can be fitted in the L2 cache.

Example 3-49. Using DCU Hardware Prefetch

Original code	Modified sequence benefit from prefetch

mov ebx, DWORD PTR [First]
xor eax, eax
scan_list:
mov eax, [ebx+4]
mov ecx, 60
 	
do_some_work_1:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_1
mov eax, [ebx+64]
mov ecx, 30
do_some_work_2:
add eax, eax
and eax, 6
sub ecx, 1
jnz do_some_work_2
	mov ebx, DWORD PTR [First]
	xor eax, eax
	scan_list:
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov eax, [ebx+4]
	mov ecx, 60
	
	do_some_work_1:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_1
	mov eax, [ebx+64]
	mov ecx, 30
	do_some_work_2:
	add eax, eax
	and eax, 6
	sub ecx, 1
	jnz do_some_work_2

mov ebx, [ebx] test ebx, ebx jnz scan_list	mov ebx, [ebx] test ebx, ebx jnz scan_list

The additional instructions to load data from one member in the modified sequence can trigger the DCU hardware prefetch mechanisms to prefetch data in the next cache line, enabling the work on the second member to complete sooner.

Software can gain from the first-level data cache prefetchers in two cases:

- If data is not in the second-level cache, the first-level data cache prefetcher enables early trigger of the second-level cache prefetcher.
- If data is in the second-level cache and not in the first-level data cache, then the first-level data cache prefetcher triggers earlier data bring-up of sequential cache line to the first-level data cache.

There are situations that software should pay attention to a potential side effect of triggering unnecessary DCU hardware prefetches. If a large data structure with many members spanning many cache lines is accessed in ways that only a few of its members are actually referenced, but there are multiple pair accesses to the same cache line.

The DCU hardware prefetcher can trigger fetching of cache lines that are not needed. In Example 3-50, references to the “Pts” array and “AltPts” will trigger DCU prefetch to fetch additional cache lines that won’t be needed. If

significant negative performance impact is detected due to DCU hardware prefetch on a portion of the code, software can try to reduce the size of that contemporaneous working set to be less than half of the L2 cache.

Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines

```
while ( CurrBond != NULL )
    {
    MyATOM  *a1 = CurrBond->At1 ;
    MyATOM  *a2 = CurrBond->At2 ;


    if ( a1->CurrStep <= a1->LastStep &&
       a2->CurrStep <= a2->LastStep
        )
          {
          a1->CurrStep++ ;
          a2->CurrStep++ ;

          double  ux = a1->Pts[0].x - a2->Pts[0].x ;
          double  uy = a1->Pts[0].y - a2->Pts[0].y ;
          double  uz = a1->Pts[0].z - a2->Pts[0].z ;
          a1->AuxPts[0].x += ux ;
          a1->AuxPts[0].y += uy ;
          a1->AuxPts[0].z += uz ;


          a2->AuxPts[0].x += ux ;
          a2->AuxPts[0].y += uy ;
          a2->AuxPts[0].z += uz ;
          } ;
    CurrBond = CurrBond->Next ;
    } ;
```

To fully benefit from these prefetchers, organize and access the data using one of the following methods:

Method 1:

- Organize the data so consecutive accesses can usually be found in the same 4-KByte page.
- Access the data in constant strides forward or backward IP Prefetcher.

Method 2:

- Organize the data in consecutive lines.
- Access the data in increasing addresses, in sequential cache lines.

Example 3-51 demonstrates accesses to sequential cache lines that can benefit from the first-level cache prefetcher.

Example 3-51. Technique for Using L1 Hardware Prefetch

```
unsigned int *p1, j, a, b;
for (j = 0; j < num; j += 16)
{
a = p1[j];
b = p1[j+1];
// Use these two values
}
```

By elevating the load operations from memory to the beginning of each iteration, it is likely that a significant part of the latency of the pair cache line transfer from memory to the second-level cache will be in parallel with the transfer of the first cache line.

The IP prefetcher uses only the lower 8 bits of the address to distinguish a specific address. If the code size of a loop is bigger than 256 bytes, two loads may appear similar in the lowest 8 bits and the IP prefetcher will be restricted. Therefore, if you have a loop bigger than 256 bytes, make sure that no two loads have the same lowest 8 bits in order to use the IP prefetcher.

3.7.3 HARDWARE PREFETCHING FOR SECOND-LEVEL CACHE

The Intel Core microarchitecture contains two second-level cache prefetchers:

- **Streamer** — Loads data or instructions from memory to the second-level cache. To use the streamer, organize the data or instructions in blocks of 128 bytes, aligned on 128 bytes. The first access to one of the two cache lines in this block while it is in memory triggers the streamer to prefetch the pair line. To software, the L2 streamer's functionality is similar to the adjacent cache line prefetch mechanism found in processors based on Intel NetBurst microarchitecture.
- **Data prefetch logic (DPL)** — DPL and L2 Streamer are triggered only by writeback memory type. They prefetch only inside page boundary (4 KBytes). Both L2 prefetchers can be triggered by software prefetch instructions and by prefetch request from DCU prefetchers. DPL can also be triggered by read for ownership (RFO) operations. The L2 Streamer can also be triggered by DPL requests for L2 cache misses.

Software can gain from organizing data both according to the instruction pointer and according to line strides. For example, for matrix calculations, columns can be prefetched by IP-based prefetches, and rows can be prefetched by DPL and the L2 streamer.

3.7.4 CACHEABILITY INSTRUCTIONS

SSE2 provides additional cacheability instructions that extend those provided in SSE. The new cacheability instructions include:

- New streaming store instructions.
- New cache line flush instruction.
- New memory fencing instructions.

For more information, see Chapter 9

3.7.5 REP PREFIX AND DATA MOVEMENT

The REP prefix is commonly used with string move instructions for memory related library functions such as MEMCPY (using **REP MOVSD**) or **MEMSET** (using **REP STOS**). These **STRING/MOV** instructions with the REP prefixes are implemented in MS-ROM and have several implementation variants with different performance levels.

The specific variant of the implementation is chosen at execution time based on data layout, alignment and the counter (**ECX**) value. For example, **MOVSB/STOSB** with the REP prefix should be used with counter value less than or equal to three for best performance.

String **MOVE/STORE** instructions have multiple data granularities. For efficient data movement, larger data granularities are preferable. This means better efficiency can be achieved by decomposing an arbitrary counter value into a number of doublewords plus single byte moves with a count value less than or equal to 3.

Because software can use SIMD data movement instructions to move 16 bytes at a time, the following paragraphs discuss general guidelines for designing and implementing high-performance library functions such as **MEMCPY()**, **MEMSET()**, and **MEMMOVE()**. Four factors are to be considered:

- **Throughput per iteration** — If two pieces of code have approximately identical path lengths, efficiency favors choosing the instruction that moves larger pieces of data per iteration. Also, smaller code size per iteration will in general reduce overhead and improve throughput. Sometimes, this may involve a comparison of the relative overhead of an iterative loop structure versus using REP prefix for iteration.
- **Address alignment** — Data movement instructions with highest throughput usually have alignment restrictions, or they operate more efficiently if the destination address is aligned to its natural data size. Specifically, 16-byte moves need to ensure the destination address is aligned to 16-byte boundaries, and 8-bytes moves perform better if the destination address is aligned to 8-byte boundaries. Frequently, moving at doubleword granularity performs better with addresses that are 8-byte aligned.
- **REP string move vs. SIMD move** — Implementing general-purpose memory functions using SIMD extensions usually requires adding some prolog code to ensure the availability of SIMD instructions, preamble code to facilitate aligned data movement requirements at runtime. Throughput comparison must also take into consideration the overhead of the prolog when considering a REP string implementation versus a SIMD approach.
- **Cache eviction** — If the amount of data to be processed by a memory routine approaches half the size of the last level on-die cache, temporal locality of the cache may suffer. Using streaming store instructions (for example: **MOVNTQ, MOVNTDQ**) can minimize the effect of flushing the cache. The threshold to start using a streaming store depends on the size of the last level cache. Determine the size using the deterministic cache parameter leaf of CPUID.
  - Techniques for using streaming stores for implementing a **MEMSET()**-type library must also consider that the application can benefit from this technique only if it has no immediate need to reference the target addresses. This assumption is easily upheld when testing a streaming-store implementation on a micro-benchmark configuration, but violated in a full-scale application situation.

When applying general heuristics to the design of general-purpose, high-performance library routines, the following guidelines can are useful when optimizing an arbitrary counter value N and address alignment. Different techniques may be necessary for optimal performance, depending on the magnitude of N:

- When N is less than some small count (where the small count threshold will vary between microarchitectures -- empirically, 8 may be a good value when optimizing for Intel NetBurst microarchitecture), each case can be coded directly without the overhead of a looping structure. For example, 11 bytes can be processed using two **MOVSD** instructions explicitly and a **MOVSB** with REP counter equaling 3.
- When N is not small but still less than some threshold value (which may vary for different micro-architectures, but can be determined empirically), an SIMD implementation using run-time CPUID and alignment prolog will likely deliver less throughput due to the overhead of the prolog. A REP string implementation should favor using a REP

string of doublewords. To improve address alignment, a small piece of prolog code using **MOVSB/STOSB** with a count less than 4 can be used to peel off the non-aligned data moves before starting to use **MOVSD/STOSD**.

- When N is less than half the size of last level cache, throughput consideration may favor either:
  - An approach using a REP string with the largest data granularity because a REP string has little overhead for loop iteration, and the branch misprediction overhead in the prolog/epilogue code to handle address alignment is amortized over many iterations.
  - An iterative approach using the instruction with largest data granularity, where the overhead for SIMD feature detection, iteration overhead, and prolog/epilogue for alignment control can be minimized. The trade-off between these approaches may depend on the microarchitecture.
  - An example of **MEMSET()** implemented using stosd for arbitrary counter value with the destination address aligned to doubleword boundary in 32-bit mode is shown in Example 3-52.
- When N is larger than half the size of the last level cache, using 16-byte granularity streaming stores with prolog/epilog for address alignment will likely be more efficient, if the destination addresses will not be referenced immediately afterwards.

Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination

A 'C' example of Memset()	Equivalent Implementation Using REP STOSD

void memset(void *dst,int c,size_t size)
{
char *d = (char *)dst;
size_t i;
for (i=0;i<size;i++)
*d++ = (char)c;
}
	push edi
	movzx eax, byte ptr [esp+12]
	mov ecx, eax
	shl ecx, 8
	or ecx, eax
	mov ecx, eax
	shl ecx, 16
	or eax, ecx
	mov edi, [esp+8] ; 4-byte aligned
	mov ecx, [esp+16] ; byte count
	shr ecx, 2 ; do dword
	cmp ecx, 127
	jle _main
	test edi, 4
	jz _main
	stosd ;peel off one dword
	dec ecx

	_main: ; 8-byte aligned rep stosd mov ecx, [esp + 16] and ecx, 3 ; do count <= 3 rep stosb ; optimal with <= 3 pop edi ret

Memory routines in the runtime library generated by Intel compilers are optimized across a wide range of address alignments, counter values, and microarchitectures. In most cases, applications should take advantage of the default memory routines provided by Intel compilers.

In some situations, the byte count of the data is known by the context (as opposed to being known by a parameter passed from a call), and one can take a simpler approach than those required for a general-purpose library routine. For example, if the byte count is also small, using **REP MOVSB/STOSB** with a count less than four can ensure good

address alignment and loop-unrolling to finish the remaining data; using **MOVSD/STOSD** can reduce the overhead associated with iteration.

Using a REP prefix with string move instructions can provide high performance in the situations described above. However, using a REP prefix with string scan instructions (**SCASB, SCASW, SCASD, SCASQ**) or compare instructions (**CMPSB, CMPSW, SMPSD, SMPSQ**) is not recommended for high performance. Consider using SIMD instructions instead.

3.7.6 ENHANCED REP MOVSB AND STOSB OPERATION

Beginning with processors based on Ivy Bridge microarchitecture, REP string operation using MOVSB and STOSB can provide both flexible and high-performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.[bit 9] = 1.

3.7.6.1 Fast Short REP MOVSB

Beginning with processors based on Ice Lake Client microarchitecture, REP MOVSB performance of short operations is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. Support for fast-short REP MOVSB is enumerated by the CPUID feature flag: CPUID [EAX=7H, ECX=0H).EDX.FAST_SHORT_REP_MOVSB[bit 4] = 1. There is no change in the REP STOS performance.

3.7.6.2 Memcpy Considerations

The interface for the standard library function memcpy introduces several factors (e.g. length, alignment of the source buffer and destination) that interact with microarchitecture to determine the performance characteristics of the implementation of the library function. Two of the common approaches to implement memcpy are driven from small code size vs. maximum throughput. The former generally uses REP MOVSD+B (see Section 3.7.5), while the latter uses SIMD instruction sets and has to deal with additional data alignment restrictions.

For processors supporting enhanced REP MOVSB/STOSB, implementing memcpy with REP MOVSB will provide even more compact benefits in code size and better throughput than using the combination of REP MOVSD+B. For processors based on Ivy Bridge microarchitecture, implementing memcpy using Enhanced REP MOVSB and STOSB might not reach the same level of throughput as using 256-bit or 128-bit AVX alternatives, depending on length and alignment factors.

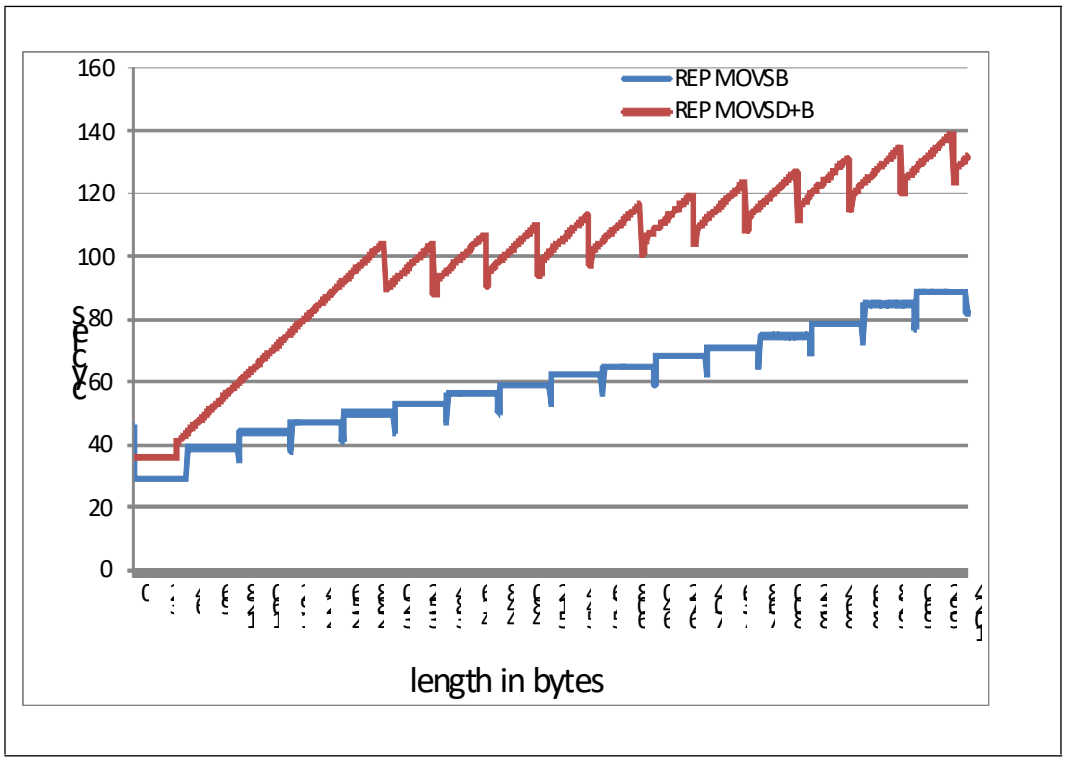


Figure 3-3. Memcpy Performance Comparison for Lengths up to 2KB

Figure 3-3 depicts the relative performance of memcpy implementation on a third-generation Intel Core processor using Enhanced REP MOVSB and STOSB versus REP MOVSD+B, for alignment conditions when both the source and destination addresses are aligned to a 16-Byte boundary and the source region does not overlap with the destination region. Using Enhanced REP MOVSB and STOSB always delivers better performance than using REP MOVSD+B. If the length is a multiple of 64, it can produce even higher performance. For example, copying 65-128 bytes takes 40 cycles, while copying 128 bytes needs only 35 cycles.

If an application wishes to bypass standard memcpy library implementation with its own custom implementation and have freedom to manage the buffer length allocation for both source and destination, it may be worthwhile to manipulate the lengths of its memory copy operation to be multiples of 64 to take advantage the code size and performance benefit of Enhanced REP MOVSB and STOSB.

The performance characteristic of implementing a general-purpose memcpy library function using a SIMD register is significantly more colorful than an equivalent implementation using a general-purpose register, depending on length, instruction set selection between SSE2, 128-bit AVX, 256-bit AVX, relative alignment of source/destination, and memory address alignment granularities/boundaries, etc.

Hence comparing performance characteristics between a memcpy using Enhanced REP MOVSB and STOSB versus a SIMD implementation is highly dependent on the particular SIMD implementation. The remainder of this section discusses the relative performance of memcpy using Enhanced REP MOVSB and STOSB versus unpublished,

optimized 128-bit AVX implementation of memcpy to illustrate the hardware capability of Ivy Bridge microarchitecture.

Table 3-5. Relative Performance of Memcpy() Using Enhanced REP MOVSB and STOSB Vs. 128-bit AVX

Range of Lengths (bytes)	<128	128 to 2048	2048 to 4096
Memcpy_ERMSB/Memcpy_AVX128	0x7X	1X	1.02X

Table 3-5 shows the relative performance of the Memcpy function implemented using enhanced REP MOVSB versus 128-bit AVX for several ranges of memcpy lengths, when both the source and destination addresses are 16-byte aligned and the source region and destination region do not overlap. For memcpy length less than 128 bytes, using Enhanced REP MOVSB and STOSB is slower than what's possible using 128-bit AVX, due to internal start-up overhead in the REP string.

For situations with address misalignment, memcpy performance will generally be reduced relative to the 16-byte alignment scenario (see Table 3-6).

Table 3-6. Effect of Address Misalignment on Memcpy() Performance

Address Misalignment	Performance Impact
Source Buffer	The impact on Enhanced REP MOVSB and STOSB implementation versus 128-bit AVX is similar.
Destination Buffer	The impact on Enhanced REP MOVSB and STOSB implementation can be 25% degradation, while 128-bit AVX implementation of memcpy may degrade only 5%, relative to 16-byte aligned scenario.

Memcpy() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit SIMD integer data-path in Haswell microarchitecture. See Section 15.16.3.

3.7.6.3 Memmove Considerations

When there is an overlap between the source and destination regions, software may need to use memmove instead of memcpy to ensure correctness. It is possible to use REP MOVSB in conjunction with the direction flag (DF) in a memmove() implementation to handle situations where the latter part of the source region overlaps with the beginning of the destination region. However, setting the DF to force REP MOVSB to copy bytes from high towards low addresses will experience significant performance degradation.

When using Enhanced REP MOVSB and STOSB to implement memmove function, one can detect the above situation and handle first the rear chunks in the source region that will be written to as part of the destination region, using REP MOVSB with the DF=0, to the non-overlapping region of the destination. After the overlapping chunks in the rear section are copied, the rest of the source region can be processed normally, also with DF=0.

3.7.6.4 Memset Considerations

The consideration of code size and throughput also applies for memset() implementations. For processors supporting Enhanced REP MOVSB and STOSB, using REP STOSB will again deliver more compact code size and significantly better performance than the combination of STOSD+B technique described in Section 3.7.5.

When the destination buffer is 16-byte aligned, memset() using Enhanced REP MOVSB and STOSB can perform better than SIMD approaches. When the destination buffer is misaligned, memset() performance using Enhanced REP MOVSB and STOSB can degrade about 20% relative to aligned case, for processors based on Ivy Bridge microarchitecture. In contrast, SIMD implementation of memset() will experience smaller degradation when the destination is misaligned.

Memset() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit data path in Haswell microarchitecture. see Section 15.16.3.3.

3.8 REP STRING OPERATIONS

Several REP string performance enhancements are available beginning with processors based on Golden Cove microarchitecture.

3.8.1 FAST ZERO LENGTH REP MOVSB

REP MOVSB performance of zero length operations is enhanced. The latency of a zero length REP MOVSB is now the same as the latency of lengths 1 to 128 bytes. When both Fast Short REP MOVSB and Fast Zero Length REP MOVSB features are enabled, REP MOVSB performance is flat 9 cycles per operation, for all strings 0-128 byte long whose source and destination operands reside in the processor first level cache.

Support for fast zero-length REP MOVSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_ZERO_LENGTH_REP_MOVSB[bit 10] = 1.

3.8.2 FAST SHORT REP STOSB

REP STOSB performance of short operations is enhanced. The enhancement applies to string lengths between 0 and 128 bytes long. When Fast Short REP STOSB feature is enabled, REP STOSB performance is flat 12 cycles per operation, for all strings 0-128 byte long whose destination operand resides in the processor first level cache.

Support for fast-short REP STOSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_STOSB[bit 11] = 1.

3.8.3 FAST SHORT REP CMPSB AND SCASB

REP CMPSB and SCASB performance is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. When the Fast Short REP CMPSB and SCASB feature is enabled, REP CMPSB and REP SCASB performance is flat 15 cycles per operation, for all strings 1-128 byte long whose two source operands reside in the processor first level cache.

Support for fast short REP CMPSB and SCASB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_CMPSB_SCASB[bit 12] = 1.

3.9 FLOATING-POINT CONSIDERATIONS

When programming floating-point applications, it is best to start with a high-level programming language such as C, C++, or Fortran. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code, the compiler may need some assistance.

3.9.1 GUIDELINES FOR OPTIMIZING FLOATING-POINT CODE

User/Source Coding Rule 9. (M impact, M generality) *Enable the compiler's use of Intel SSE, Intel SSE2, Intel AVX, Intel AVX2, and possibly more advanced SIMD instruction sets (Intel AVX-512) with appropriate switches. Favor scalar SIMD code generation to replace x87 code generation.*

Follow this procedure to investigate the performance of your floating-point application:

- Understand how the compiler handles floating-point code.
- Look at the assembly dump and see what transforms are already performed on the program.
- Study the loop nests in the application that dominate the execution time.
- Determine why the compiler is not creating the fastest code.
- See if there is a dependence that can be resolved.
- Determine the problem area: bus bandwidth, cache locality, trace cache bandwidth, or instruction latency. Focus on optimizing the problem area. For example, adding PREFETCH instructions will not help if the bus is already saturated. If trace cache bandwidth is the problem, added prefetch µops may degrade performance.

Also, in general, follow the general coding recommendations discussed in this chapter, including:

- Blocking the cache.
- Using prefetch.
- Enabling vectorization.
- Unrolling loops.

User/Source Coding Rule 10. (H impact, ML generality) *Make sure your application stays in range to avoid denormal values, underflows.*

Out-of-range numbers cause very high overhead.

When converting floating-point values to 16-bit, 32-bit, or 64-bit integers using truncation, the instructions **CVTTSS2SI** and **CVTTSD2SI** are recommended over instructions that access x87 FPU stack. This avoids changing the rounding mode.

User/Source Coding Rule 11. (M impact, ML generality) *Usually, math libraries take advantage of the transcendental instructions (for example, FSIN) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider an alternate, software-based approach, such as a look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision and the size of the look-up table, and by taking advantage of the parallelism of the Intel SSE and the Intel SSE2 instructions.*

3.9.2 FLOATING-POINT MODES AND EXCEPTIONS

When working with floating-point numbers, high-speed microprocessors frequently must deal with situations that need special handling in hardware or code.

3.9.2.1 Floating-Point Exceptions

The most frequent cause of performance degradation is the use of masked floating-point exception conditions such as:

- Arithmetic overflow.
- Arithmetic underflow.
- Denormalized operand.

Refer to Chapter 4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for definitions of overflow, underflow and denormal exceptions.

Denormalized floating-point numbers impact performance in two ways:

- Directly when are used as operands.
- Indirectly when are produced as a result of an underflow situation.

If a floating-point application never underflows, the denormals can only come from floating-point constants.

User/Source Coding Rule 12. (H impact, ML generality) *Denormalized floating-point constants should be avoided as much as possible.*

Denormal and arithmetic underflow exceptions can occur during the execution of x87 instructions or Intel SSE/Intel SSE2/Intel SSE3 instructions. Processors based on Intel NetBurst microarchitecture handle these exceptions more efficiently when executing Intel SSE/Intel SSE2/Intel SSE3 instructions and when speed is more important than complying with the IEEE standard. The following paragraphs give recommendations on how to optimize your code to reduce performance degradations related to floating-point exceptions.

3.9.2.2 Dealing with Floating-Point Exceptions in x87 FPU Code

Every special situation listed in Section 3.9.2.1 is costly in terms of performance. For that reason, x87 FPU code should be written to avoid these situations.

There are basically three ways to reduce the impact of overflow/underflow situations with x87 FPU code:

- Choose floating-point data types that are large enough to accommodate results without generating arithmetic overflow and underflow exceptions.
- Scale the range of operands/results to reduce as much as possible the number of arithmetic overflow/underflow situations.
- Keep intermediate results on the x87 FPU register stack until the final results have been computed and stored in memory. Overflow or underflow is less likely to happen when intermediate results are kept in the x87 FPU stack (this is because data on the stack is stored in double extended-precision format and overflow/underflow conditions are detected accordingly).
- Denormalized floating-point constants (which are read-only, and hence never change) should be avoided and replaced, if possible, with zeros of the same sign.

3.9.2.3 Floating-Point Exceptions in SSE/SSE2/SSE3 Code

Most special situations that involve masked floating-point exceptions are handled efficiently in hardware. When a masked overflow exception occurs while executing Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 code, processor hardware can handles it without performance penalty.

Underflow exceptions and denormalized source operands are usually treated according to the IEEE 754 specification[1], but this can incur significant performance delay. If a programmer is willing to trade pure IEEE 754 compliance for speed, two non-IEEE 754 compliant modes are provided to speed situations where underflows and input are frequent: FTZ mode and DAZ mode.

When the FTZ mode is enabled, an underflow result is automatically converted to a zero with the correct sign. Although this behavior is not compliant with IEEE 754, it is provided for use in applications where performance is more important than IEEE 754 compliance. Since denormal results are not produced when the FTZ mode is enabled,

1. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229.

the only denormal floating-point numbers that can be encountered in FTZ mode are the ones specified as constants (read only).

The DAZ mode is provided to handle denormal source operands efficiently when running a SIMD floating-point application. When the DAZ mode is enabled, input denormals are treated as zeros with the same sign. Enabling the DAZ mode is the way to deal with denormal floating-point constants when performance is the objective.

If departing from the IEEE 754 specification is acceptable and performance is critical, run Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 applications with FTZ and DAZ modes enabled.

NOTE

The DAZ mode is available with both the Intel SSE and Intel SSE2 extensions, although the speed improvement expected from this mode is fully realized only in SSE code and later.

3.9.3 FLOATING-POINT MODES

For x87 code, using the FLDCW instruction to change floating modes can be an expensive operation in many cases.

Recent processor generations provide hardware optimization for **FLDCW** that allows programmers to alternate between two constant values efficiently. For the **FLDCW** optimization to be effective, the two constant FCW values are only allowed to differ on the following 5 bits in the FCW:

FCW[8-9] ; Precision control
FCW[10-11] ; Rounding control
FCW[12] ; Infinity control

If programmers need to modify other bits (for example: mask bits) in the FCW, the **FLDCW** instruction is still an expensive operation.

In situations where an application cycles between three (or more) constant values, **FLDCW** optimization does not apply, and the performance degradation occurs for each **FLDCW** instruction.

One solution to this problem is to choose two constant FCW values, take advantage of the optimization of the **FLDCW** instruction to alternate between only these two constant FCW values, and devise some means to accomplish the task that requires the 3rd FCW value without actually changing the FCW to a third constant value. An alternative solution is to structure the code so that, for periods of time, the application alternates between only two constant FCW values. When the application later alternates between a pair of different FCW values, the performance degradation occurs only during the transition.

It is expected that SIMD applications are unlikely to alternate between FTZ and DAZ mode values. Consequently, the SIMD control word does not have the short latencies that the floating-point control register does. A read of the **MXCSR** register has a fairly long latency, and a write to the register is a serializing instruction.

There is no separate control word for single and double precision; both use the same modes. Notably, this applies to both FTZ and DAZ modes.

Assembly/Compiler Coding Rule 52. (H impact, M generality) *Minimize changes to bits 8-12 of the floating-point control word. Changes for more than two values (each value being a combination of the following bits: precision, rounding and infinity control, and the rest of bits in FCW) leads to delays that are on the order of the pipeline depth.*

3.9.3.1 Rounding Mode

Many libraries provide float-to-integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. With the Pentium 4 processor, one can use the **CVTTSD2SI** and **CVTTSS2SI** instructions to convert operands with truncation

without ever needing to change rounding modes. The cost savings of using these instructions over the methods below is enough to justify using Intel SSE and Intel SSE2 wherever possible when truncation is involved.

For x87 floating-point, the **FIST** instruction uses the rounding mode represented in the floating-point control word (FCW). The rounding mode is generally "round to nearest", so many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the **FLDCW** instruction. For a change in the rounding, precision, and infinity bits, use the **FSTCW** instruction to store the floating-point control word. Then use the **FLDCW** instruction to change the rounding mode to truncation.

In a typical code sequence that changes the rounding mode in the FCW, a **FSTCW** instruction is usually followed by a load operation. The load operation from memory should be a 16-bit operand to prevent store-forwarding problem. If the load operation on the previously-stored FCW word involves either an 8-bit or a 32-bit operand, this will cause a store-forwarding problem due to mismatch of the size of the data between the store operation and the load operation.

To avoid store-forwarding problems, make sure that the write and read to the FCW are both 16-bit operations.

If there is more than one change to the rounding, precision, and infinity bits, and the rounding mode is not important to the result, use the algorithm in Example 3-53 to avoid synchronization issues, the overhead of the **FLDCW** instruction, and having to change the rounding mode. Note that the example suffers from a store-forwarding problem which will lead to a performance penalty. However, its performance is still better than changing the rounding, precision, and infinity bits among more than two values.

Example 3-53. Algorithm to Avoid Changing Rounding Mode

```
_fto132proc
    lea       ecx, [esp-8]
    sub       esp, 16             ; Allocate frame
    and       ecx, -8             ; Align pointer on boundary of 8
    fld       st(0)               ; Duplicate FPU stack top

    fistp     qword ptr[ecx]
    fild      qword ptr[ecx]
    mov       edx, [ecx+4]        ; High DWORD of integer
    mov       eax, [ecx]          ; Low DWIRD of integer
    test      eax, eax
    je        integer_QnaN_or_zero
arg_is_not_integer_QnaN:
    fsubp     st(1), st           ; TOS=d-round(d), { st(1) = st(1)-st & pop ST}
    test      edx, edx            ; What's sign of integer
    jns       positive            ; Number is negative
    fstp      dword ptr[ecx]      ; Result of subtraction
    mov       ecx, [ecx]          ; DWORD of diff(single-precision)
    add       esp, 16
    xor       ecx, 80000000h
    add       ecx,7fffffffh       ; If diff<0 then decrement integer
    adc       eax,0               ; INC EAX (add CARRY flag)
    ret
positive:
```

Example 3-53. Algorithm to Avoid Changing Rounding Mode (Contd.)

```
    positive:
    fstp    dword ptr[ecx]    ; 17-18 result of subtraction
    mov     ecx, [ecx]        ; DWORD of diff(single precision)
    add     esp, 16
    add     ecx, 7fffffffh    ; If diff<0 then decrement integer
    sbb     eax, 0            ; DEC EAX (subtract CARRY flag)
    ret
integer_QnaN_or_zero:
    test    edx, 7fffffffh
    jnz     arg_is_not_integer_QnaN
    add esp, 16
    ret
```

Assembly/Compiler Coding Rule 53. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision, and infinity bits.*

3.9.3.2 Precision

If single precision is adequate, use it instead of double precision. This is true because:

- Single precision operations allow the use of longer SIMD vectors, since more single precision data elements can fit in a register.
- If the precision control (PC) field in the x87 FPU control word is set to single precision, the floating-point divider can complete a single-precision computation much faster than either a double-precision computation or an extended double-precision computation. If the PC field is set to double precision, this will enable those x87 FPU operations on double-precision data to complete faster than extended double-precision computation. These characteristics affect computations including floating-point divide and square root.

Assembly/Compiler Coding Rule 54. (H impact, L generality) *Minimize the number of changes to the precision mode.*

3.9.4 X87 VS. SCALAR SIMD FLOATING-POINT TRADE-OFFS

There are a number of differences between x87 floating-point code and scalar floating-point code (using Intel SSE and Intel SSE2). The following differences should drive decisions about which registers and instructions to use:

- When an input operand for a SIMD floating-point instruction contains values that are less than the representable range of the data type, a denormal exception occurs. This causes a significant performance penalty. An SIMD floating-point operation has a flush-to-zero mode in which the results will not underflow. Therefore subsequent computation will not face the performance penalty of handling denormal input operands. For example, in the case of 3D applications with low lighting levels, using flush-to-zero mode can improve performance by as much as 50% for applications with large numbers of underflows.
- Scalar floating-point SIMD instructions have lower latencies than equivalent x87 instructions. Scalar SIMD floating-point multiply instruction may be pipelined, while x87 multiply instruction is not.
- Although x87 supports transcendental instructions, software library implementation of transcendental function can be faster in many cases.
- x87 supports 80-bit precision, double extended floating-point. SSE support a maximum of 32-bit precision. SSE2 supports a maximum of 64-bit precision.
- Scalar floating-point registers may be accessed directly, avoiding **FXCH** and top-of-stack restrictions.

- The cost of converting from floating-point to integer with truncation is significantly lower with Intel SSE and Intel SSE2 in the processors based on Intel NetBurst microarchitecture than with either changes to the rounding mode or the sequence prescribed in the Example 3-53.

Assembly/Compiler Coding Rule 55. (M impact, M generality) *Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack.*

3.9.4.1 Scalar Intel® SSE/Intel® SSE2

In code sequences that have conversions from floating-point to integer, divide single-precision instructions, or any precision change, x87 code generation from a compiler typically writes data to memory in single-precision and reads it again in order to reduce precision. Using Intel SSE/Intel SSE2 scalar code instead of x87 code can generate a large performance benefit using Intel NetBurst microarchitecture and a modest benefit on Intel Core Solo and Intel Core Duo processors.

Recommendation: Use the compiler switch to generate scalar floating-point code using XMM rather than x87 code.

When working with Intel SSE/Intel SSE2 scalar code, pay attention to the need for clearing the content of unused slots in an XMM register and the associated performance impact. For example, loading data from memory with MOVSS or MOVSD causes an extra micro-op for zeroing the upper part of the XMM register.

3.9.4.2 Transcendental Functions

If an application needs to emulate math functions in software for performance or other reasons (see Section 3.9.1), it may be worthwhile to inline math library calls because the **CALL** and the prologue/epilogue involved with such calls can significantly affect the latency of operations.

3.10 MAXIMIZING PCIE PERFORMANCE

PCIe performance can be dramatically impacted by the size and alignment of upstream reads and writes (read and write transactions issued from a PCIe agent to the host's memory). As a general rule, the best performance, in terms of both bandwidth and latency, is obtained by aligning the start addresses of upstream reads and writes on 64-byte boundaries and ensuring that the request size is a multiple of 64-bytes, with modest further increases in bandwidth when larger multiples (128, 192, 256 bytes) are employed. In particular, a partial write will cause a delay for the following request (read or write).

A second rule is to avoid multiple concurrently outstanding accesses to a single cache line. This can result in a conflict which in turn can cause serialization of accesses that would otherwise be pipelined, resulting in higher latency and/or lower bandwidth. Patterns that violate this rule include sequential accesses (reads or writes) that are not a multiple of 64-bytes, as well as explicit accesses to the same cache line address. Overlapping requests—those with different start addresses but with request lengths that result in overlap of the requests—can have the same effect. For example, a 96-byte read of address 0x00000200 followed by a 64-byte read of address 0x00000240 will cause a conflict—and a likely delay— for the second read.

Upstream writes that are a multiple of 64-byte but are non-aligned will have the performance of a series of partial and full sequential writes. For example, a write of length 128-byte to address 0x00000070 will perform similarly to 3 sequential writes of lengths 16, 64, and 48 to addresses 0x00000070, 0x00000080, and 0x00000100, respectively.

For PCIe cards implementing multi-function devices, such as dual or quad port network interface cards (NICs) or dual-GPU graphics cards, it is important to note that non-optimal behavior by one of those devices can impact the bandwidth and/or latency observed by the other devices on that card. With respect to the behavior described in this section, all traffic on a given PCIe port is treated as if it originated from a single device and function.

For the best PCIe bandwidth:

- Align start addresses of upstream reads and writes on 64-byte boundaries.
- Use read and write requests that are a multiple of 64-bytes.
- Eliminate or avoid sequential and random partial line upstream writes.
- Eliminate or avoid conflicting upstream reads, including sequential partial line reads.

Techniques for avoiding performance pitfalls include cache line aligning all descriptors and data buffers, padding descriptors that are written upstream to 64-byte alignment, buffering incoming data to achieve larger upstream write payloads, allocating data structures intended for sequential reading by the PCIe device in such a way as to enable use of (multiple of) 64-byte reads. The negative impact of unoptimized reads and writes depends on the specific workload and the microarchitecture on which the product is based.

3.10.1 OPTIMIZING PCIE PERFORMANCE FOR ACCESSES TOWARD COHERENT MEMORY AND MMIO REGIONS (P2P)

In order to maximize performance for PCIe devices in the processors listed in Table 3-7 the software should determine whether the accesses are toward coherent (system) memory or toward MMIO regions (P2P access to other devices). If the access is toward MMIO region, then software can command HW to set the RO bit in the TLP header, as this would allow hardware to achieve maximum throughput for these types of accesses. For accesses toward coherent memory, software can command HW to clear the RO bit in the TLP header (no RO), as this would allow hardware to achieve maximum throughput for these types of accesses.

Table 3-7. Intel Processor CPU RP Device IDs for Processors Optimizing PCIe Performance

Processor	CPU RP Device IDs
Intel® Xeon® processors based on Broadwell microarchitecture	6F01H-6F0EH
Intel® Xeon® processors based on Haswell microarchitecture	2F01H-2F0EH

3.11 SCALABILITY WITH CONTENDED LINE ACCESS IN 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

A two-socket system as found in the Sapphire Rapids microarchitecture can have up to 224 (2 sockets x 56 cores/socket x 2 threads/core) hardware threads. Scalability and performance bottlenecks may happen when all of these hardware threads compete for the same address.

3.11.1 CAUSES OF PERFORMANCE BOTTLENECKS

When multiple hardware threads go after the same address (for example, AA), this address is queued in the Ingress Queue, with one entry for each hardware thread. Due to the resource limitation of the Ingress Queue, the CPU core is throttled to slow the rate of requests when this queue overflows. This usually occurs with contention for a lock.

3.11.2 PERFORMANCE BOTTLENECK DETECTION

When multiple cores are contending on the same lock, several outstanding requests are mapped to that same address. The Phys_addr_match event can count as such an event. This CHA event increments by one every other cycle when there is more than one outstanding request to the same address.

Here are the PMU event id and Umask for the 2 CHA events that are very useful for detecting contention:

- **Phys_addr_match event:** Event id: 0x19, Umask: 0x80
- **CHA_clockticks event:** Event id: 0x01, Umask: 0x01

These events have to be measured on a per-CHA basis, and if the ratio of the counts between phys_addr_match to CHA_clockticks is more than 0.15 on any CHA that indicates > 30% of the CHA cycles (2x the ratio as this event can count only once every two cycles) are spent with multiple requests outstanding to the same address.

The Recipe to Measure Events with Linux perf:

```
$ sudo perf stat -a -e
'uncore_cha/event=0x19,umask=0x80/,uncore_cha/event=0x1,umask=0x1/' --per-socket
--no-merge -- sleep 30
```

Once confirmed that the ratio of phys_addr_match events to the CHA clockticks is more than 0.15, the next step is figuring out where this may be happening in the code. Intel CPUs provide a PMU mechanism wherein a load operation is randomly selected and tracked through completion, and the true latency is recorded if it is over a given threshold. The threshold value is specified in cycles and must be in the power of 2. In the following “perf mem record” command, define a command to sample all loads that take more than 128 cycles to complete.

```
$ sudo perf mem record -a --ldlat 128 sleep 1
```

Once the above data is collected, execute the following command to process the data collected:

```
$ sudo perf mem report
```

Information similar to the table below will be generated. Such information will include details on hot loads along with data linear address and the actual latency that the load experienced. This can be used to identify the necessary fixes to the code.

Table 3-8. (Sheet 1 of 2)Samples: 365K of Events ‘anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}’, Event Count (a--r0x): 67900852

Overhead	Samples	Local Weight	Memory Access	Symbol	Shared Object	Data Symbol	Data Object	Snoop	TLB Access	Locked	Blocked	Local INSTR Latency
0.22% 0.07%	1	1 38060	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	47251

Table 3-8. (Contd.)(Sheet 2 of 2)Samples: 365K of Events 'anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}', Event Count (a--r0x): 67900852

Overhead	Samples	Local Weight	Memory Access	Symbol	Shared Object	Data Symbol	Data Object	Snoop	TLB Access	Locked	Blocked	Local INSTR Latency
0.18% 0.06%	1	1 31338	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	40411
0.17% 0.06%	1	1 29572	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	36652

3.11.3 SOLUTIONS FOR PERFORMANCE BOTTLENECKS

The following is a list of suggested solutions:

- Run multiple instances of the workload with a scale-out approach instead of a single instance with scale-up so that the contention for per instance hot variables (including locks) is reduced.
- Guard the cmpxchg by checking that the destination memory is expected with a load, test, and branch beforehand.
- Implement a backoff mechanism so that the cmpxchg is issued less. For example, in locks, exponential backoff is a common and effective method to prevent all cores from being in lockstep. In the case of contention for a lock, checking to see if it is accessible by a load before trying to write to it through a cmpxchg will help.

The code in Example 3-54 provides an example:

Example 3-54. Locking Algorithm for the Sapphire Rapids Microarchitecture

```
        lock_loop:
        while (lock is not free) // just a load operation
        execute pause;

// now the lock is free, so try to acquire it.
        Exponential Backoff spin // so all the cores don't come back at the same time
        Execute cmpxchg on the lock
        if the lock is not successfully acquired, goto lock_loop
```

Additionally, as the core counts continue to increase, exploring other algorithmic fixes that dissolve or reduce contention on memory variables (including locks) is essential. For example, instead of frequently updating a hot statistical variable from all threads, consider updating a copy of it per thread (without contention) and later aggregate the updated per-thread copies on a less frequent basis or use some existing atomic-free concurrency methods such as rseq[1]. As another example, restructure locking algorithms to use hierarchical locking when excessive contention is detected on a global lock.

3.11.4 CASE STUDY: SYSBENCH/MARIADB

With SysBench/MariaDB 10.3.34[2], the workload's throughput drops as the number of threads increases. Another metric we can use is the CHA% Cycles Fast Asserted. It is a signal to slow down the cores when the Ingress Queue fills up. This is another way to identify scalability issues. The graph below plots the number of active client threads representing the work intensity on the horizontal axis. The percentage of Fast Asserts is plotted on the vertical axis.

The baseline case (blue line) had a sharp throughput with increased thread count, as all cores reduced their throughput as they suffered from the increasing percent of Fast Asserts. With the same work distributed instances (red line), Fast asserts dropped. Similarly, with a software fix (gray line), again, the Fast Asserts dropped even though only one instance was in execution.

1. https://git.kernel.org/pub/scm/libs/librseq/librseq.git/tree/doc/man/rseq.2
2. The most current version is MariaDB 10.3.39

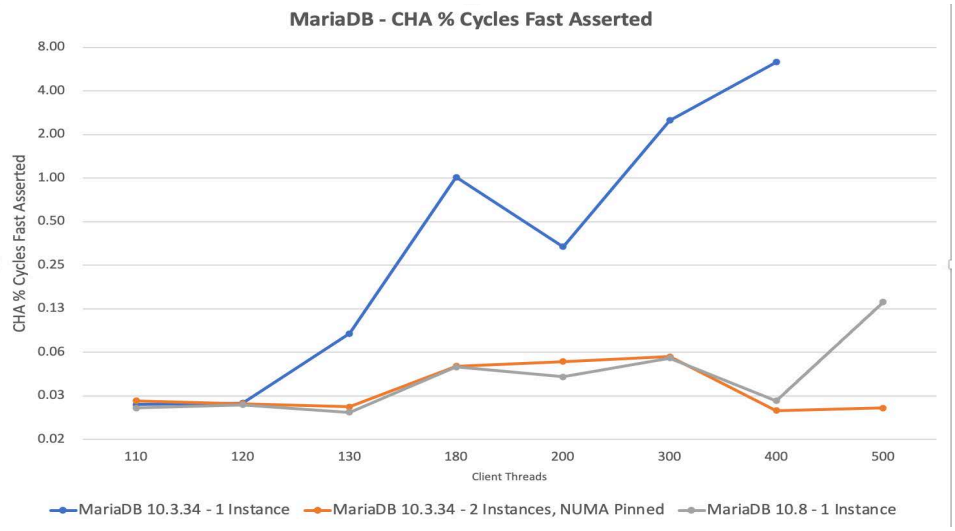


Figure 3-4. MariaDB - CHA % Cycles Fast Asserted

3.11.5 SCALABILITY WITH FALSE SHARING

A two-socket 4th Generation Intel® Xeon® Scalable Processors 8480 system can support up to 224 hardware threads (2 sockets x 56 cores per socket x 2 threads per core). However, when multiple threads concurrently access different variables in a structure that happen to reside in the same cache line, it can result in false sharing leading to scalability issues. False sharing can cause unnecessary cache invalidations and updates leading to significant performance degradation in multi-threaded programs that utilize all the hardware threads. Therefore, it is essential to avoid false sharing by designing data structures and memory layouts that minimize contention on shared cache lines to achieve optimal performance in multi-threaded environments.

3.11.5.1 Causes of False Sharing

False sharing is a performance problem that can occur in multi-threaded programming when threads access different variables sharing the same cache line. Cache lines are units of memory that are loaded into the processor's cache. When multiple threads write different variables in the same cache line, they end up competing for access to the cache line. This results in cache invalidations and updates that are unnecessary, which can lead to a significant performance degradation. This problem gets worse when many threads are contending for the same cache line.

3.11.5.2 Detecting False Sharing

The perf c2c is a profiling tool available in Linux that detects false sharing issues by analyzing cache-to-cache (c2c) transfers between threads. It works by intercepting the cache coherence messages sent between threads and identifying the specific cache lines that are involved in false sharing. The perf c2c approach generates a report that shows the amount of time spent on c2c transfers, the number of bytes transferred, and the specific cache lines that are affected by false sharing. This approach provides a more precise and accurate method of detecting false sharing issues compared to traditional profiling tools, as it directly measures the cache coherence overhead caused by false sharing. The perf c2c approach is particularly useful for detecting subtle false sharing issues that may not be visible using other profiling tools.

Hardware Invalidation Tracking Modified (HITM) is a counter in the perf c2c output that represents the number of cache lines that were modified in one cache and then invalidated in another cache due to both false and true sharing. The HITM counter provides insight into the performance impact of false sharing by measuring the number of

unnecessary cache invalidations and the resulting traffic between caches. By reducing false sharing, the HITM counter can be reduced, leading to better performance and scalability in multi-threaded programs.

Steps for perf c2c analysis[1]:

1. Collect perf c2c data on the target system (this example is for the full system):

   "perf c2c record -a -u --ldlat 50 -- sleep 30

2. Generate report (this can take considerable time to process)

   "perf c2c report -NN -g --call-graph --full-symbols -c pid,iaddr --stdio >perf_report.txt

3. Check the generated perf_report.txt for "Shared Data Cache Line Table" (see Table 3-9). This table is sorted by the HITM. Pay attention to the topped "CacheLine address". See Example 3-55.

4. Read the perf_report.txt for the "Shared Cache Line Distribution Pareto" (see Table 3-10). Check the "Offset" column to see if there are multiple offset within single cache line. If there are multiple offset, that points to a potential false s haring issue. See Example 3-56.

3.11.5.3 Fixing False Sharing and Additional Resources

The following is a list of suggested solutions:

1. Add padding so the fields are not on the same cache line. Example 3-56 shows to prevent the false sharing between the **full** and **empty lfstack** variables padding is added between them. This is the fix detailed in Section 3.11.5.4. This has the additional effect of increasing the memory sizes and may create other false sharing for other variables.

2. Run multiple instances of the workload instead of a single instance so that the false sharing for per in-stance false sharing variables is reduced as fewer hardware threads are allocated per instances.

3. Change other parameters to prevent the false sharing. In the case of Go, the GOGC variable can be tuned to reduce this.

4. In some environments, it may not be desirable to increase data structure sizes. In this case there may be other patterns to follow such as splitting up a data structure or changing writes for some global variable to use compare(read)-then-write instead of unconditional write. However, this will require further code refactoring.

The Linux kernel has documented some kernel specific False Sharing issues and how to mitigate them.

A blog by a Netflix engineer details how they used a variety of tools including the Intel PMCs (Performance Monitoring Counters) to find and fix False Sharing in JVM.

Example 3-55. Perf Annotation for runtime.getempty

```
next :* atomic.Load64(&node.next)
6290    425fb6: mov     (%rcx) ,%rdx              // lfstack.go.48
        425fb9 lea      0x87fb88(%rip) ,%rbx
9703    425fc0 lock     cmpxchng %rdx,(%rbx)      // lfstack.go.49
        425fc5 sete     %dl
        425fc8 test     %dl,%dl
        425fca je       425f9e <runtime.getempty+0x19e>
        425fcc jmp      425fde <runtime.getempty+0x1d0>
        425fce xor      %ecx,%ecx
```

1. The blog, https://joemario.github.io/blog/2016/09/01/c2c-blog/ , provides a nice introduction to perf c2c in Linux.

Example 3-56. Padding Insertion in Go Runtime

```
src/runtime/mgc.go
@@ -285,8 + @@ func pollFractionalWorkerExit() bool {

var work struct {
    full l-stack                          // lock-free list of full blocks workbuf.
+ pad0 cpu.CacheLinePad                   // prevents false-sharing between full and empty.
    empty l-stack                         // lock-free list of empty blocks workbuf.
```

3.11.5.4 Case Study: DeathStarBench/hotelReservation

DeathStarBench is an open-source benchmark suite for microservice workloads, originally developed by Cornell University. It represents different applications written in modern cloud native architecture. The hotelReservation workload in DeathStarBench mimics a typical microservice workload: a hotel booking system. It is written in Golang and uses gRPC-go for inter-microservice communication.

When running with the default parameters and on a single instance of the workload, perf c2c shows false sharing issue with the DSB HR workload. Table 3-10 shows that there are two different offsets being modified by different threads/functions for the specific cache line.

Table 3-9. Shared Data Cache Line Table

Cache Line				Total Hitm	Load Hitm	
Index	Address	Node	PA cnt		Total	LclHitm
0	**0xca5b40**	1	19364	3.25%	9083	9083
1	**0xd9a840**	0	10918	1.66%	4652	4652
2	**0xce1140**	1	10613	1.56%	4352	4352
3	**0xd9a080**	0	8300	1.14%	3181	3181
4	**0xd9a8c0**	0	4274	0.87%	2448	2448
5	**0xd95900**	0	5346	0.83%	2334	2334
6	**0xd9d800**	1	5440	0.83%	2324	2324
7	**0xce0980**	1	6129	0.83%	2319	2319
8	**0xd98800**	1	5117	0.77%	2160	2160

Table 3-10. Shared Cache Line Distribution Pareto (Sheet 1 of 2)

HTTM		Data Address						
RmtHitm	LclHitm	Offset	Node	Total Records	cpu cnt	Symbol	Shared Object	Souce:Line
0.00%	15.26%	0x0	1	8272	112	[.] runtime.gcDrainN	frontend	mgmark.go:1186
0.00%	8.99%	0x8	1	7276	112	[.] runtime.gcDrain	frontend	mgmark.go:1028

Table 3-10. Shared Cache Line Distribution Pareto (Contd.)(Sheet 2 of 2)

HTTM		Data Address						
RmtHitm	LclHitm	Offset	Node	Total Records	cpu cnt	Symbol	Shared Object	Souce:Line
0.00%	7.99%	0x8	1	2850	112	[.] runtime.trygetfull	frontend	lfstack.go.49
0.00%	3.36%	0x8	1	2827	112	[.] runtime.trygetfull	frontend	mgcwork.go.421
0.00%	3.01%	0x8	1	1324	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.35
0.00%	1.94%	0x0	1	885	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.33
0.00%	37.84%	0x8	1	9239	112	[.] runtime.getempty	frontend	lfstack.go.49
0.00%	6.90%	0x8	1	2875	112	[.] runtime.(*lfstack).push	frontend	fstack.go.35
0.00%	5.92%	0x8	1	7188	112	[.] runtime.getempty	frontend	lfstack.go.43
0.00%	4.81%	0x8	1	1947	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.33
0.00%	2.87%	0x8	1	1012	112	[.] runtime.getempty	frontend	mgcwork.go.350

To find the root causes, perf annotate target function:

perf annotate --tui -l -n "runtime.(*lfstack).push

and review the source code to identify false sharing. In this case the update of full and empty **lfstack** variables by hardware threads on different cores causes the false sharing.

After identifying and fixing the false sharing problem in the Golang runtime (it is in the Go Runtime), releasing in and recompiling the workload binary with the modified Golang runtime improved the throughput metric by 12%. As the following table shows, other metrics such as the CPI and the CHA Fast Asserts also improve significantly. The perf c2c report also shows no additional false sharing.

Table 3-11. False Sharing Improvements

Metric	False Sharing Fix/Base
TPS	1.12
CPI	0.84
Metric CHA % cycles Fast Asserted	0.42

3.11.6 INSTRUCTION SEQUENCE SLOWDOWNS

The Golden Cove CPU microarchitecture upon which the Sapphire Rapids microarchitecture is based has increased the cost of mixing Legacy SSE and VEX without clearing the state of upper registers for power efficiency reasons.

3.11.6.1 Causes of Instruction Sequence Slowdowns

The Golden Cove CPU microarchitecture eliminated some hardware speed paths for power efficiency and replaced them with microcode. The instruction sequence in Table 3-12 mixes VEX and Legacy SSE. It has, for example, higher

core cycles than on the previous generation Sunny Cove CPU microarchitecture for the Ice Lake version of the 3rd Generation of Intel Xeon Scalable processors. The higher core cycles are due to the execution of additional micro-operations.

Table 3-12. Instruction Sequence Mixing VEX on the Sapphire Rapids and Ice Lake Server Microarchitectures

Intel Assembly Code Syntax	Ice lake Server Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)	
VPXOR XMM3, XMM3, XMM3; VEXTRACTI128 XMM3, YMM3, 1; PXOR XMM3, XMM3	Inst Retired	Core Cycles	Inst Retired	Core Cycles
	3.00	1	3.00	388.04

3.11.6.2 Detecting Instruction Sequence Slowdowns

The event ASSISTS.SSE_AVX_MIX can be used to determine if there are VEX to legacy SSE transitions. The following Linux perf command-line can be used while the workload is running:

$ sudo perf stat -e 'assists.sse_avx_mix'[1] <workload>

With the Intel® TMA (Topdown Methodology) (there is a metric called Mixing_Vectors which gives the percentage of injected blend µops out of all the µops issued. Usually, a Mixing_Vectors metric over 5% is worth investigating. You can find more details in Appendix B1 of the Optimizations Guide.

3.11.6.3 Fixing Instruction Sequence Slowdowns

The following is a list of suggested solutions:

1. When possible, use VEX-encoded instructions for all the SIMD instructions when possible.
2. Insert a VZEROUPPER to tell the hardware that the state of the higher registers is clean between the VEX and the legacy SSE instructions.
   a. Often the best way to do this is to insert a VZEROUPPER before returning from any function that uses VEX (that does not produce a VEX register) and before any call to an unknown function.
3. VZEROUPPER was inserted in the code sequence below and there are no SSE_AVX_MIX assists.

1. Using upstream perf. If OS doesn't have support for the event use cpu/event=0xc1,umask=0x10,name=assists_sse_avx_mix/

a. With this change, the Core Cycles do not have a performance inversion relative to the previous generation.

Table 3-13. Fixed Instruction Sequence with Improved Performance on Sapphire Rapids Microarchitecture

	Ice lake Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)		ASSISTS.SSE_AVX_MIX
Intel Assembly Code Syntax	**Inst Retired**	**Core Cycles**	**Inst Retired**	**Core Cycles**	**ASSISTS.SSE_AVX_MIX**
VPXOR XMM3, XMM3, XMM3; VEXTRACTI128 XMM3, YMM3, 1; PXOR XMM3, XMM3	4.00	2.00	4.00	1.00	0

3.11.7 MISPREDICTION FOR BRANCHES >2GB

The Golden Cove CPU is a wider machine and might exhibit a higher Top-down Microarchitecture Analysis (TMA) Bad Speculation percentage. See Section 22.1.6 for additional information about TMA. Some sources of Bad Speculation are branch prediction misses. In this case, however, Bad Speculation is due to the wider machine and less efficient branch prediction for certain indirect branches.

3.11.7.1 Causes of Branch Misprediction >2GB

For a near absolute indirect JMP/CALL branch instruction (opcodes FF /4 and FF /2), the branch distance (ADDR_TARGET - ADDR_BRANCH) affects the performance of the branch predictor. The branch predictor uses fewer resources to predict the branch if its distance can be specified with a 32-bit signed displacement (JMP/CALL imm32). If the distance is larger (>2GB), the predictor uses more resources to predict the branch and performance may suffer.

3.11.7.2 Detecting Branch Mispredictions >2GB

You can use the Last Branch Record (LBR) to identify jumps greater than 2GB. The collection of performance analysis tools based on perf on Linux supports this. The following is an example output from the tool. It shows that 21% of the call/jumps of >2GB offset are mispredicted. The histogram of one of the indirect branches at address 0x555555603664 shows that it is to one target and in a library. The profile mask is to use LBR, and the duration is 10 seconds. It does a system-level profile.

Example 3-57. Identifying >2GB Branches

```
% ./do.py profile --profile-mask=0x100 -s 10

count of indirect call/jump of >2GB offset: 93200
count of mispredicted indirect call/jump of >2GB offset: 19943
misprediction ratio for indirect branch at address 0x7ffff577eca4: 4.23%
misprediction ratio for indirect branch at address 0x5555556030c4: 32.23%
misprediction ratio for indirect branch at address 0x555555603664: 22.30%
misprediction ratio for indirect branch at address 0x555555603c24: 13.84%
…
indirect_0x555555603664 histogram:
0x7ffff7af2670:  50501 100.0%
```

3.11.7.3 Fixing Branch Mispredictions >2GB

Arrange the code so the jumps don't span the >2GB range. This can be done through a variety of approaches:

1. If possible, statically link all the libraries into the executable.
2. For **.text** to library code, use the Glibc environment variable LD_PREFER_MAP_32BIT_EXEC=1 to restrict the addresses into the 4GB range.
3. For dynamically compiled code, keep it close to the .text address or copy the frequently called entries into the dynamically compiled code address region[1].

In a case study with WordPress/PHP running eight containers with and without the 2GB fix, the CPI and performance scores improve by 6%.

Table 3-14. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction

	WP4.2 / PHP7.4.29 - NO FIX	WP4.2 / PHP7.4.29 - 2G FIX in Glibc	2G FIX/NO FIX
Config (Workers)	8c x 42	8c x 42	-
Config (Cores Per socket)	56	56	1.00
Config (Sockets)	2	2	1.00
Config (Total Cores)	112	112	1.00
Config (Total Thread Count)	224	224	1.00
Performance Throughput	1.00	1.06	1.06
Performance CPI	1.12	1.05	0.96
Path Length (Instructions/Unit of Work)	33,789,862.68	33,730,155.10	1.00
Cycles/Transaction (Per Unit of Work)	37,803,310.48	35,359,628.33	0.94

1. See the Google V8 Blog.

3.12 OPTIMIZING COMMUNICATION WITH PCI DEVICES ON INTEL® 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

The Sapphire Rapids microarchitecture introduced a new set of instructions designed to optimize communication between SW running on IA cores and PCI devices on the platform.

3.12.1 SIGNALING DEVICES WITH DIRECT MOVE

Most software-to-device interaction follows a producer-to-consumer relationship where the software creates work for the device and then signals it to inform the device that work is available. Descriptor rings are the ubiquitous pattern here and once descriptors are added to the ring, the signal (or "doorbell") consists of an update to the tail pointer register on the device. This is a write to an MMIO-mapped BAR register.

Such writes tend to be relatively expensive operations –the latency to complete the write to the device is high relative to the CPU operating speed. Since writes are ordered by default, this creates a bubble during which subsequent writes cannot be drained from store buffers. Signaling can therefore affect performance via store backpressure.

As a result, some software libraries avoid frequent signaling by batching relatively large quantities of work descriptors with each doorbell update. However, this is not always possible, and it introduces latency.

The Sapphire Rapids microarchitecture introduces "Direct Store" instructions to optimize signaling; there are two instructions in the family:

- MOVDIRI: 4/8B direct store.
- MOVDIR64B: 64B atomic direct copy.

Direct Stores are weakly ordered (like non-temporal or USWC-mapped memory writes) regardless of the underlying memory type (which is usually UC for MMIO-mapped locations). Since they do not order subsequent writes the performance issue described above does not occur.

Since they are intended for signaling, direct stores will never combine with other stores to the same address as can happen with non-temporal or USWC writes. Each write is guaranteed to occur as issued. In the case of MOVDIR64B, the full 64B will be delivered as a single write to the device. This is the only ISA that carries an architectural guarantee of >8B atomicity.

These instructions benefit from the fact that signaling use cases typically do not care if subsequent writes are observed before the doorbell itself because the ordering is relaxed. However, since typically the doorbell must not be observable before earlier writes (such writes are creating the work descriptors), SW should insert a store fence immediately before the direct store.

Having a fence before the direct store does not normally limit performance– except when many direct stores are issued. If there is an SFENCE before each, the fence on direct store N+1 imposes an order on direct store N, which can remove some of the benefits. The guideline is to avoid this where possible. One technique that may work if multiple doorbells to different addresses are being issued (such as for a NIC driver that is handling multiple descriptor rings), is to group the direct stores to different locations together and insert a single SFENCE before the group.

Notably, the device write latency can vary widely with the address being written. This is especially true on large CPUs implemented as multiple tiles. So if SW has the luxury of choosing between multiple addresses, it is possible to envisage adaptive schemes that "match" an address to a SW thread (especially if that thread is pinned to a single core) by selecting the best performing such address during an initialization stage.

3.12.1.1 MOVDIR64B: Additional Considerations

As noted above MOVDIR64B is a copy operation; it moves data from one 64B-aligned address to another. Typical usage is that the source address is a memory location, and the destination is MMIO mapped to a device, whereupon it confers the benefits described above. However, since the source data is usually written immediately before the MOVDIR64B, additional considerations include:

- It is unnecessary to fence to ensure the source data is written before the MOVDIR64B since the source data is written to the same address that the MOVDIR64B reads. In some scenarios, no store fence is needed in conjunction with MOVDIR64B. The correct operation of the system depends on being observed before the MOVDIR64B if no other data is written to memory.
- It is critical to allow store forwarding of the source data for the best performance.
- The source data should be aligned to 64B and written at the same granularity that the MOVDIR64B reads. For the Sapphire Rapids microarchitecture, this is 64B: the source data should, therefore, be written using 64B Intel® AVX-512 Instructions for the best performance.

3.12.1.2 Streaming Data

MOVDIR64B can also be used to stream data to a device by copying a block of memory because it is weakly ordered. This is similar behavior to mapping the destination memory locations as USWC, except:

- The destination address can remain mapped UC.
- The writes are guaranteed to arrive at the device as 64B writes, which is not guaranteed with any other method.

3.13 SYNCHRONIZATION

3.13.1 USER-LEVEL MONITOR, USER-LEVEL MWAIT, AND TPAUSE

New instructions for user-level monitor and MWAIT act like legacy monitor and MWAIT instructions with additional functionality identified as the timeout and ring-3 (user space) application support. TPAUSE is similar to legacy pause instruction but is designed to accept time interval and sleep state parameters. User-level MWAIT and TPAUSE support the same C0.1 light sleep and C0.2 deeper sleep states. These instructions are helpful in user space applications that support a busy poll, synchronization, or asynchronous IO, such as waiting for an event. A minor code modification yields power benefits along with low latency wake-up.

3.13.1.1 Checking for User-Level Monitor, MWAIT, and TPAUSE Support

This section describes how to check whether a processor supports user-level monitor, user-level MWAIT, or TPAUSE; if user-level monitor, user-level MWAIT, or TPAUSE instruction is supported, then CPUID. (EAX=07H, ECX=0): ECX [bit 5] is enumerated as 1.

Example 3-58. Identification of WAITPKG with CPUID

```
…identify the existence of cpuid instruction
… ;
… ;
Identify signature is genuine Intel …;
mov eax, 7; Request for feature flags
mov ecx, 0; Request for feature flags
cpuid; 0FH, A2H CPUID instruction
test ecx, 00000020h;
Is waitpkg bit (bit 5) in feature flags equal to 1 jnz Found
```

3.13.1.2 User-Level Monitor, User-Level MWAIT, and TPAUSE Operations

User-level monitor initializes the monitor hardware in such a way that, after execution of the user-level MWAIT, a store to a monitored address acts as a wakeup event. So, the User level monitor and the user-level MWAIT work together to obtain a sleep state. TPAUSE is a single instruction request to enter one of the same two sleep states for a defined time

There are possibilities of a “false wake-up” because of other events, notably interrupts or timeouts. The application may re-execute user-level MWAIT/TPAUSE if it has been falsely woken. If the application needs to determine the source of the predefined OS sleep wakeup, RFLAGS.CF is set Otherwise it is assumed that the application can detect changes at the monitored address (MWAIT) or poll for activity (TPAUSE).

3.13.1.3 Recommended Usage of Monitor, MWAIT, and TPAUSE Operations

A frequent paradigm in packet processing applications is to have dedicated HW threads polling a NIC receive descriptor ring for ingress traffic. This kind of “busy polling” arrangement wastes energy when the traffic rates are low. Changing the polling loop to perform user-level Monitor/MWAIT on the next descriptor to be written can save substantial power in periods of low traffic. The same scheme could be used with any “work distributor,” assigning work by writing to selected memory locations.

Accelerators frequently offload tasks from SW in an asynchronous manner. For example, the Intel® Data Streaming Accelerator (Intel® DSA) performs copy operations and can return the status of the completed operation by writing to memory. If an application uses the user-level monitor/MWAIT at a memory location where the status field will be written, it can be woken when the task is complete. Instead of monitoring, the device may issue an interrupt that can act as a wake-up event.

Alternatively, applications may decide to choose TPAUSE as a wait event. This has the advantage of being independent of the number of event sources.

In all cases, a small change in the user space application is needed to convert a busy poll application to something more energy efficient with low latency wake-up.

Synchronous application

- when two hardware threads from the same core use user-level monitor and user-level MWAIT, it can progress effectively as some of the hardware resources are available to the other thread when a hyperthread issues the user-level MWAITs.

To achieve the best performance using user-level monitor and user-level MWAIT:

- The entire contents of monitored locations must be verified after user-level MWAIT to avoid a false wake-up.
- It is the developer's responsibility to check the contents of monitored locations:
  - Before issuing monitor.
  - Before issuing user-level MWAIT.
  - After user-level MWAIT. See Example 3-59.
- If an application expects a store to a monitored location, the timeout value should be as high as it is supported.

Since user-level MWAIT and TPAUSE are a hint to a processor, a user should selectively identify locations in the application.

Example 3-59. Code Snippet in an Asynchronous Example

```
void * m_address;  // it is expected device will update m_address to 1
unsigned char ret;
while (1) {
  if (*m_address != 0) // if device already finished operation, no need to user monitor/user mwait
     break;
  if (*m_address == 0) { // check monitored location before issuing umonitor instruction
     _umonitor (m_address);
     if (*m_address == 0) {        // check monitored location before issuing umwait instruction
        ret = _umwait(0, 0x186A0);    // some high value in timeout
     }
  }
}
```

CHAPTER 4
INTEL ATOM® PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel Atom® processors.

4.1 THE CRESTMONT MICROARCHITECTURE

The Crestmont microarchitecture builds on the success of the Gracemont microarchitecture. Listed below are some of the many enhancements provided by the Gracemont microarchitecture.

- Increased Branch Prediction Bandwidth (128B/cycle max from 32B/cycle on Gracemont).
- Larger Branch Target Buffer (6K entry from 5K) with Enhanced Path Based Branch Prediction.
- Wider allocation width (6-wide from 5-wide).
- Larger second-level TLB and larger dedicated 1GB page TLB.
- 48-bit VA with 52-bit PA used for MKTME keys.
- 2x SIMD integer multiply units, faster integer divide units.
- VEX-based AVX-NE-CONVERT convert, AVX-VNNI-INT8 and AVX-IFMA ISA extension.
- ECC protected Data Cache (in server products).
- Linear address masking (LAM)[1], Linear Address Space Separation (LASS)[2], Secure Arbitration Mode (SEAM), and Trust Domain Extensions (TDX) ISA extensions.
- Performance Monitoring enhancements include eight general-purpose counters (from six), precise distribution support for general-purpose counter 1 (totaling three counters), timed PEBS support, LBR event logging support, and multiple new events.

1. Not available in the Meteor Lake microarchitecture.
2. *ibid*.

4.1.1 CRESTMONT MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Crestmont microarchitecture is depicted in Figure 4-1.

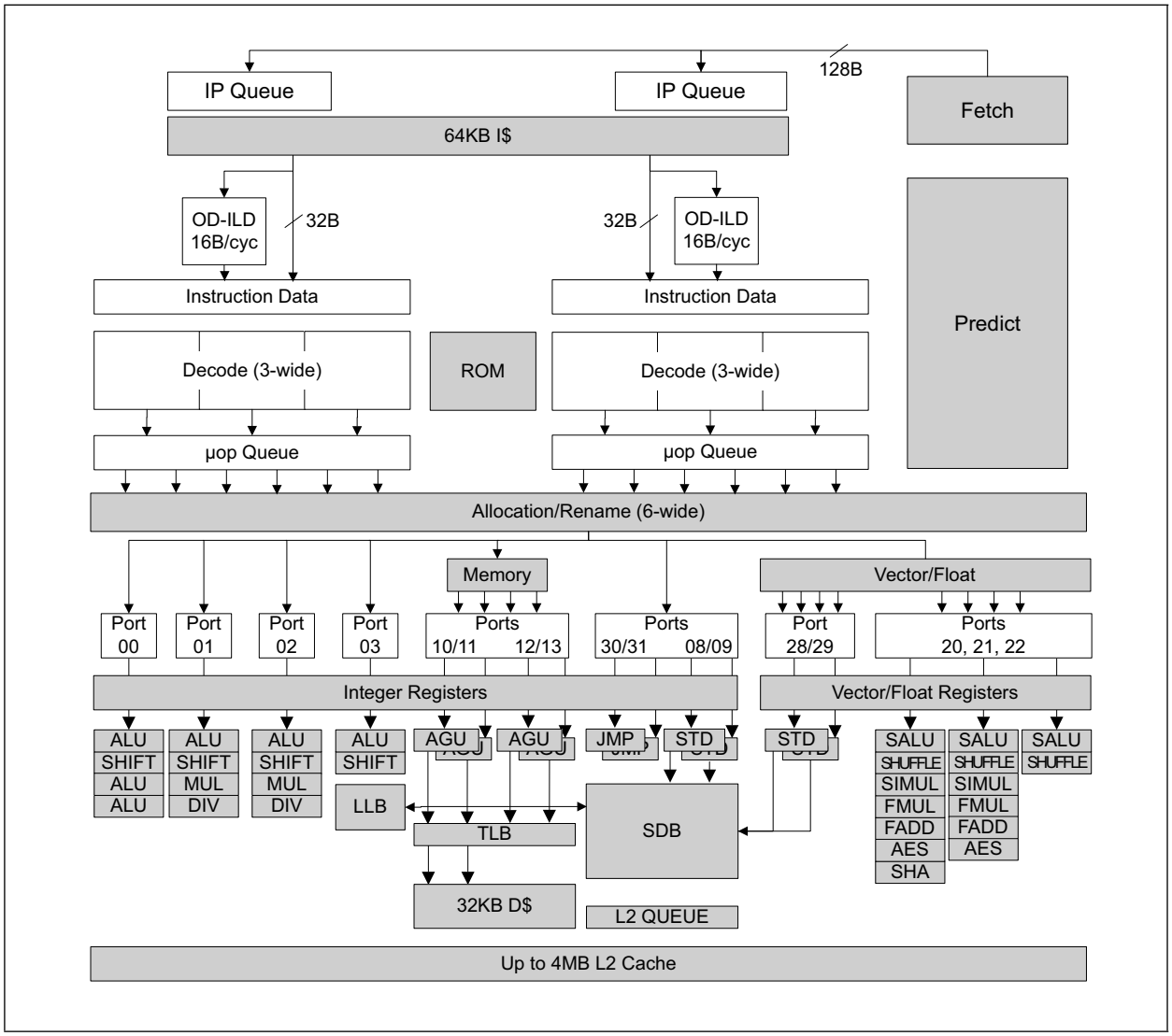


Figure 4-1. Processor Core Pipeline Functionality of the Crestmont Microarchitecture

The Crestmont microarchitecture supports flexible integration of multiple processor cores with a shared un-core subsystem consisting of a number of components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

4.1.2 PREDICT AND FETCH

Crestmont features a front end with up to 128 bytes per cycle prediction. This is dramatically larger than the 32B/cycle from the Gracemont generation. To process all 128 bytes, the address must be aligned such that an even 64B line plus an odd 64B line can be processed together based on bit 7 of the linear address. If there is a taken branch to an odd 64B line, that prediction cycle will process only the 64B line. 128B/cycle processing will resume the following cycle, assuming no taken branches were detected.

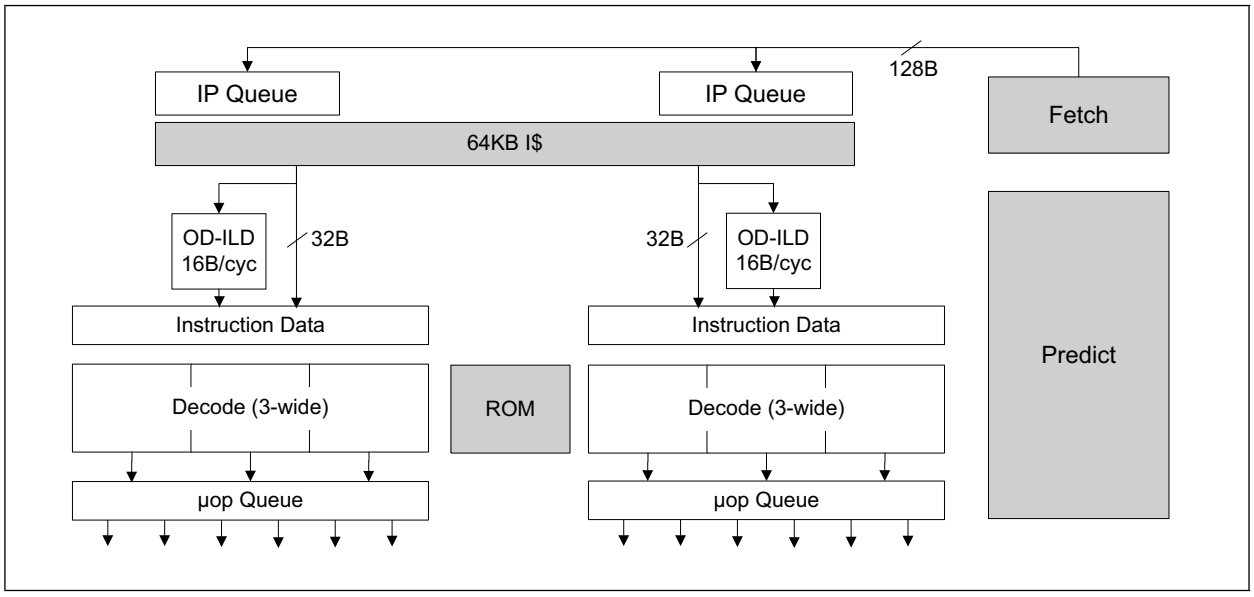


Figure 4-2. Front-End Pipeline Functionality of the Crestmont Microarchitecture

Each cycle, the predicted IP is sent down the instruction fetch pipeline. These predictions can look up the Instruction TLB (ITLB) and the instruction cache tag to determine the physical address and instruction cache hit or miss. Upon successful translation, and depending on resource availability, these accesses are stored into the instruction pointer (IP) queues. This enables the decoupling instruction cache hit/miss from delivering raw instruction bytes to the rest of the front end. In the case of an instruction cache miss, the IP queue holds the address but signals that the data cannot be read until it is returned from the memory subsystem. The stream of IPs generated at fetch can handle up to eight concurrent instruction cache misses. There are two independent IP queues, each with its instruction data buffers. Combined with their associated decoders, these are referred to as clusters. For each taken branch or inserted toggle point, the prediction will toggle back and forth between each IP queue and cluster. This toggling enables out-of-order decode, which is the key feature that enables this microarchitecture to fetch and decode up to 6 variable length x86 instructions per cycle.

Performance debug of prediction or fetch can be done utilizing the front-end bound events in the top-down category of performance monitoring events[1]. Front-end bound events count slots at allocation only when slots are available, but no µops are present. If bubbles caused by the three-cycle predictor percolate to allocation, for example, these will be represented by TOPDOWN_FE_BOUND.BRANCH_RESTEER. You can precisely tag the instruction following such a bubble via FRONTEND_RETIRED.BRANCH_RESTEER. If the predictor failed to cache a branch target and redirection occurred during decode, those slots are counted by TOPDOWN_FE_BOUND.BRANCH_DETECT. If µops are not delivered due to misses in the Instruction Cache or Instruction TLB, these appear as TOPDOWN_FE_BOUND.ICACHE and TOPDOWN_FE_BOUND.ITLB, respectively. Like BRANCH_RESTEER, all front-end bound slot-based accounting can be tracked precisely via the corresponding FRONTEND_RETIRED set of events. The instruction code can often be

1. Please see https://perfmon-events.intel.com.

rearranged to optimize such a bottleneck away. Multiple event classes can be tracked simultaneously (e.g., mark both ICACHE and ITLB events) on the same general-purpose performance counter or with different events across multiple performance counters.

Sometimes, a code loop is too short and/or poorly aligned within the cache to enable the machine to decode sufficiently fast. In this situation you could be fetching every cycle and never inserting bubbles, but still unable to keep the back-end fed. When this happens, the event class that detects this is TOPDOWN_FE_BOUND.OTHER. The “other” event class catches front-end bound behavior that cannot be pinpointed to other specific sources.

4.1.3 DYNAMIC LOAD BALANCING

Because Crestmont increased the prediction rate in many cases, it is expected to report fewer FE_BOUND.OTHER cases compared to Gracemont in general. However, since Crestmont also increased the allocation width by 20%, there are now more SLOTS that must be accounted for. Additional toggle points can be created based on internal heuristics when the hardware detects long basic blocks. These toggle points are added to the predictors, guiding the machine to toggle within the basic block and keeping both decode clusters busy.

4.1.4 INSTRUCTION DECODE AND THE ON-DEMAND INSTRUCTION LENGTH DECODER (OD-ILD)

Crestmont maintains the Gracemont on-demand instruction length decoding blocks. These blocks are typically only active when new instruction bytes are brought into the instruction cache due to a miss. When this happens, two extra cycles are added to the fetch pipeline of the affected decode cluster to generate pre-decode bits on the fly. Each block process is done across 16 bytes per cycle. With clustering, this means Crestmont is capable of 32 bytes per cycle across the two independent OD-ILDs (as in Gracemont).

Like Gracemont, Crestmont has two 3-wide instruction decode clusters capable of 6 instruction decode per cycle. Each instruction decoder generates a single µop yet can generate most x86 code as measured by dynamic instruction count. Load-op-stores, complicated addressing forms, Control Enforcement Technology (CET) instructions, and many more types are generated in a single internal µop format. Each decoder is also capable of detecting a microcode entry point. The most common short microcode flows can be executed out of order between the clusters, enabling additional performance. All µops are written into two parallel µop queues (one queue per 3-wide cluster), designed to allow the front and back end of the core to execute independently. The allocation and rename pipeline reads both µop queues in parallel and returns the instruction stream for register renaming and resource allocation.

The low-level characteristics of the microarchitecture within each decode cluster have remained identical since the Tremont microarchitecture[1].

If you are doing performance debugging and think load balancing or other decode restrictions may be an issue, this will often be indicated by TOPDOWN_FE_BOUND.DECODE. If the decoder struggled due to not having the correct pre-decode bits OR there were too many prefixes or escapes on the instructions, this would be represented by TOPDOWN_FE_BOUND.PREDECODE. If the machine is stuck waiting on lengthy microcode sequences, this will be represented by TOPDOWN_FE_BOUND.CISC. As with all other allocation slot-based FE_BOUND events, there are corresponding FRONTEND_RETIRED events that mark an instruction **after** the designated event class occurred. However, there is a difference in how this is reported for CISC events. As slot-based bottlenecks due to executing long microcode instructions are typically seen “within” an instruction, FRONTEND_RETIRED.CISC will often tag the CISC instruction itself, not the following instruction. When microcode is invoked to handle external interrupts, faults, traps, or other types of assists, FRONTEND_RETIRED.CISC will mark the next instruction that follows.

1. Refer to Chapter 6, “Earlier Generations of Intel Atom® Microarchitecture and Software Optimization” in the Intel® 64 and IA-32 Architectures Optimization Reference Manual Documentation Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency.

4.1.5 ALLOCATION AND RETIREMENT

Crestmont can allocate up to six µops per cycle, compared to five µops in Gracemont. Allocation reads the µop queues of all front-end clusters simultaneously and generates an in-order stream splicing across clustering boundaries within the same cycle as necessary. In some cases, there can be an expansion between the format inside the µop queue and the format allocated to the machine. For example, for a 256-bit AVX instruction, the front-end decodes the instruction as a single µop, subdivided into 128-bit operations at allocation time. In this case, two allocation lanes allocate the two 128-bit halves of the instruction. The most common µops that use this method besides 256-bit AVX µops are integer µops that require multiple logical register destinations, like integer multiplies and divides. Another example is PUSH memory, which loads a value from memory from one address, stores the value into memory at the location of the stack pointer, and updates the stack pointer. MOV elimination, NOP detection, idiom detection (for example, XOR, a register by itself, producing all zeros), and memory renaming are performed at allocation time. This can reduce dependency chains and, in some situations, eliminate µops from execution.

For the 256-entry retirement buffer, retirement can be up to eight instructions per cycle. Retirement is wider than allocation to improve performance for store deallocation and other less common flushing conditions.

4.1.6 THE OUT-OF-ORDER AND EXECUTION ENGINES

The Out-of-Order and execution engines in the Crestmont microarchitecture remain mostly unchanged from the Gracemont microarchitecture with enhancements in:

- Integer divide units.
- Double SIMD integer multiply units supporting VNNI and new INT8 VNNI for 2x throughput.
- Support for AVX-IFMA in FMUL unit.

The execution pipeline functionality of the Crestmont microarchitecture is shown in Figure 4-3.

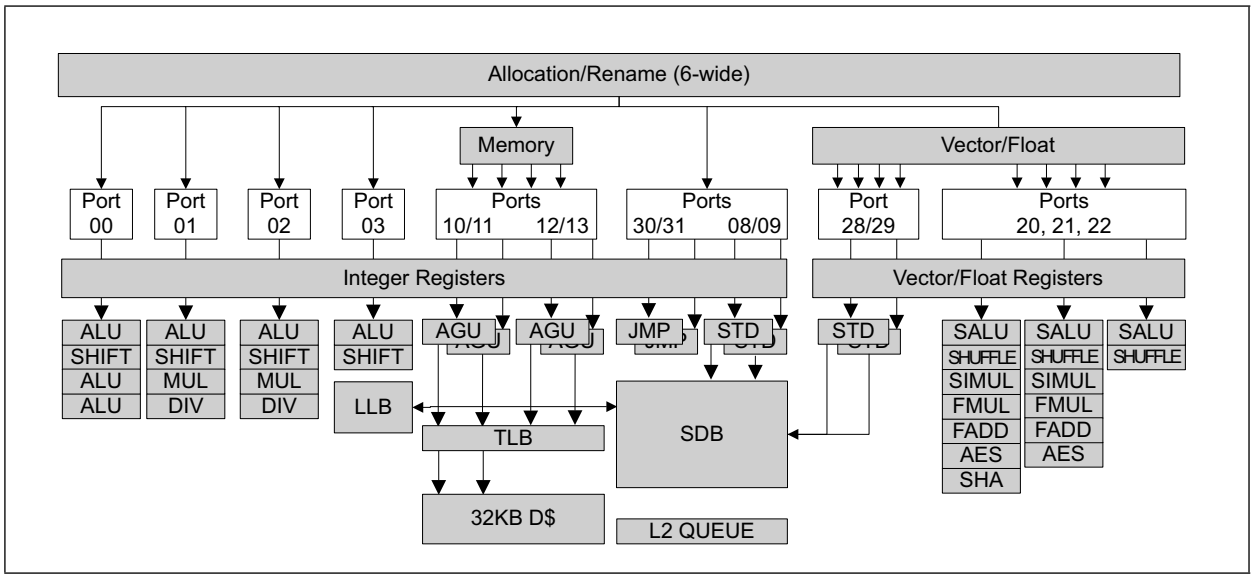


Figure 4-3. Execution Pipeline Functionality of the Crestmont Microarchitecture

Allocation delivers µops to three types of structures. Each µop is written into one or more of five reservation stations for pure integer operations. These hold instructions, track their dependencies, and schedule them for execution.

- Four are for ALU operations, labeled ports 00 to 03.

  - These execution units are mostly symmetric for single-cycle operations.
- Two of the four ports (01 and 02) can execute longer latency operations like multiplies and divides.
- The fifth integer reservation station holds *jump* and *store data* operations.
  - This structure is banked and can schedule two µops of each type every cycle
    - Two store data on ports 08 and 09.
    - Two jumps on ports 30 and 31.
- Complex instructions like an ADD, where one source and the destination are in memory, are decoded by the front end and allocated as a single µop.

The Crestmont microarchitecture can allocate five to six instructions like these per cycle. However, such µops separate into multiple pieces as they enter the back end. In this example, this single complex µop generates:

- A load.
- An add.
- A store address operation.
- A store data operation.

These pieces execute independently in the out-of-order machine, requiring four different dispatch ports.

Load Effective Address Operations (LEAs) are special and deserve extra attention.

The ALU ports are optimized to execute standard two-source arithmetic/logical operations, while the AGUs are optimized to handle the complexities of x86 memory addressing.

- LEAs are ALU operations that can have the same complex characteristics as AGU operations.
- LEAs without a scaled index and with only two sources among base, index, and displacement execute as a normal ALU operation on any port (00 through 03).
- LEAs with three sources fracture into two operations and take an additional cycle of latency.
- LEAs with a scaled index without a displacement execute as a single operation but are statically bound to port 02.

Allocation can also write into a memory queue.

This FIFO queue enables deeper buffering of the microarchitecture at a very low implementation cost. The memory queue can then write into a unified reservation station that holds load and store address generation operations. This reservation station can generate two load (ports 10 and 11) and store address calculations (ports 12 and 13)per cycle. The memory queue also writes the load and stores µops into the memory subsystem to perform translation and data cache access.

Finally, allocation can write the vector queue. This is where all vector SIMD and floating-point ALU operations go. This FIFO queue can then write into a unified reservation with three scheduling pipelines (ports 20, 21, and 22) or a store data reservation station capable of dispatching two store data per cycle (ports 28 and 29). The vector unit can execute any combination of two floating-point multiplies, adds, or multiply-add operations. This enables a peak of sixteen single-precision or eight double-precision FLOPS per cycle. It can also execute up to three SIMD integer ALU or shuffle operations along with dedicated AES and SHA units.

4.1.7 CACHE AND MEMORY SUBSYSTEM

The cache and memory subsystem remain largely unchanged from Gracemont other than the following enhancements:

- Enhanced store-to-load forwarding to cover more partial forwarding conditions.
- Larger second-level TLB.

- Larger and dedicated 1GB page TLB.
- Faster handling of locked loads.
- Support for Page Modification Logging and LAM.
- Faster eviction protocol.
- Dead Block Prediction to optimize last-level cache usage in some SoCs.
- Server SKU-specific features, including support for “end-to-end” parity and ECC in the L1 data cache.
- New performance monitoring events.
- Enhancements to the Memory Bandwidth Enforcement (MBE) QoS feature.

The Crestmont microarchitecture’s memory subsystem is designed to handle two 16 byte loads and two 16 byte stores per cycle, providing simultaneous 32 bytes of read bandwidth and 32 bytes of write bandwidth per cycle. The load-to-use latency for loads is typically four cycles. Suppose you are doing a pointer-chasing operation where the computed address results from a single prior load and a positive displacement of no more than +1023. In that case, the load-to-use latency observed can be reduced to three cycles. The L1 data cache is dual-ported to eliminate potential bank conflicts.

Memory disambiguation is supported, which allows loads to execute while older stores have unresolved addresses. Loads that forward from stores can do so in the same load to use latency as cache hits for cases where the store’s address is known, and the store data is available. Precise blocking and scheduling are done for cases where the store address or data is not immediately available and the hardware has determined that these are likely to be related addresses.

Address translations are performed through the first level DTLB, which is fully associative. On Crestmont, 2MB translations are natively cached within the first-level DTLB. The DTLB is backed by two second-level TLB (STLB) structures shared between code and data requests. The main STLB is 3072 entries 6-way set associative and caches 4KB and 2MB translations. Additionally, Crestmont has a 16-entry fully associative structure for 1 GB translations. STLB misses are sent to the page miss handler (PMH) which is pipelined such that it can perform up to four walks in parallel.

Table 4-1. Paging Cache Parameters of the Crestmont Microarchitecture

Level	Entries	Associativity	Architectural Page Size	Cached Translation Size
ITLB	64	Fully associative	All	4KB, 256KB
DTLB	32	Fully associative	All	4KB, 2MB
STLB	3072	6-Way	4K/2M/4M	4KB, 2MB
STLB	16	Fully associative	1GB	1GB

There are three independent L1 prefetchers. One does a simple next-line fetch on DL1 load misses. An instruction pointer-based prefetcher capable of detecting striding access patterns of various sizes. This prefetcher works in the linear address space; it can, therefore, cross page boundaries and start translations for TLB misses. The final prefetcher is a next-page prefetcher that detects accesses likely to cross a page boundary and starts the access early. L1 data misses generated by these prefetchers communicate additional information to the L2 prefetchers, which helps them work together.

The L2 cache delivers 64 bytes of data per cycle at a latency of 17 cycles, and that bandwidth is shared amongst 4 cores. The L2 cache subsystem also contains multiple prefetchers, including a streaming prefetcher that detects striding access patterns. An additional L2 prefetcher attempts to detect more complicated access patterns. These

prefetches can also be generated such that they only fill the LLC but do not fill into the L2 to help reduce DRAM latency.

The L2 cache subsystem of a single 4-core module can have 64 requests and 32 L2 data evictions outstanding on the fabric. To ensure fairness, these are competitively shared amongst the cores with per-core reservations.

4.1.8 CRESTMONT NEW INSTRUCTION SUPPORT

4.1.8.1 AVX-NE-CONVERT Instructions

The AVX-NE-CONVERT includes an instruction set that converts low-precision floating points like BF16/FP16 to high-precision floating point FP32. It can also convert FP32 elements to BF16. These instructions allow the platform to have improved AI capabilities and better compatibility. Crestmont implements BF16/FP16 up-conversion in the load pipelines, which do not consume FPC execution resources. This allows for low latency conversions with simultaneous floating point compute.

4.1.8.2 AVX-IFMA

AVX-IFMA includes two instructions, VPMADD52LUQ and VPMADD52HUQ. They are designed to accelerate Big Integer Arithmetic (BIA). These instructions can multiply eight 52-bit unsigned integers residing in YMM registers, produce the low (VPMADD52LUQ) and high (VPMADD52HUQ) halves of the 104-bit products, and add the results to 64-bit accumulators (i.e., SIMD elements), placing them in the destination register.

4.1.8.3 AVX-VNNI-INT8 Instructions

For more flexibility within a convolutional neural network for INT8 software, add all signed/unsigned INT8 data type support combinations to VNNI.

4.1.9 LEGACY INTEL® AVX1/INTEL® AVX2 INSTRUCTION SUPPORT

The Crestmont microarchitecture continues to support Intel® AVX and Intel® AVX2 instructions.

Most 256-bit Intel AVX and Intel AVX2 instructions are decoded as a single instruction and stored as a single µop in the front-end pipeline. To execute 256-bit instructions on native 128-bit vector execution and load data paths, most 256-bit µops are further subdivided into two independent 128-bit µops at allocation before insertion into the MEC and FPC reservation stations. These two independent µops are usually assigned to different execution ports, so both may execute in parallel. In general, 256-bit µops consume twice the allocation, execution, and retirement resources compared to 128-bit µops.

While most 256-bit Intel AVX2 instructions can be decomposed into two independent 128-bit micro-operations, a subset of Intel AVX2 instructions, known as cross-lane operations, can only compute the result for an element by utilizing one or more sources belonging to other elements. For example, when some or all of the upper 128-bit result [255:128] depends upon one or all of a lower element segment [127:0]. For example, when some or all of the upper 128-bit result [255:128] depends on one or all of a lower element segment [127:0]. These 256-bit cross-lane instructions execute with longer latency and/or reduced throughput than their 256-bit non-cross-lane counterparts.

4.1.9.1 256-bit Permute Operations

The instructions listed below use more operand sources than can be natively supported by a single reservation station within these microarchitectures. They are decomposed into two µops, where the first µop resolves a subset of

operand dependencies across two cycles. The dependent second µop executes the 256-bit operation by using a single 128-bit execution port for two consecutive cycles with a five-cycle latency for a total latency of seven cycles.

- **VPERM2I128** ymm1, ymm2, ymm3/m256, imm8
- **VPERM2F128** ymm1, ymm2, ymm3/m256, imm8
- **VPERMPD** ymm1, ymm2/m256, imm8
- **VPERMPS** ymm1, ymm2, ymm3/m256
- **VPERMD** ymm1, ymm2, ymm3/m256
- **VPERMQ** ymm1, ymm2/m256, imm8

4.1.9.2 256-bit Broadcast with 128-bit Memory Operand

The memory versions of the broadcast instructions listed below have a single 128-bit or less memory source operand, a single SIMD ALU µop, and a load operand. The register version of the same instructions is decomposed into two SIMD ALU µops.

Operation portion latency is one cycle in addition to load operation latency.

- **VBROADCASTSD** ymm1, m64
- **VBROADCASTSS** ymm1, m32

4.1.9.3 256-bit Insertion, Up-Conversion Instructions with 128-bit Memory Operand

The memory versions of the instructions listed below have a single 128-bit or less memory source operand. They are decomposed into two µops. However, the second micro-operation for the memory version depends on the first micro-operation, while the second micro-operation of the register version of the same instruction does not. The register version of the same instructions can execute the upper and lower 128-bit segments in parallel.

Operation portion latency is two cycles in addition to load operation latency for the 256-bit insert, packed move with zero, and sign extension instructions listed below.

- **VPMOVZX** ymm1, m128/64/32
- **VPMOVSX** ymm1, m128/64/32
- **VINSERTI128** ymm1, ymm2, m128, imm8
- **VINSERTF128** ymm1, ymm2, m128, imm8

Operation portion latency is six cycles in addition to load operation latency for the up-convert instructions listed below.

- **VCVTPS2PD** ymm1, m128
- **VCVTDQ2PD** ymm1, m128
- **VCVTPH2PS** ymm1, m128

4.1.9.4 256-bit Variable Blend Instructions

The **VBLENDVPD** and **VBLENDVPS** instructions listed below are implemented as a micro-coded flow. Throughput is one every four cycles, and latency is three cycles.

- **VBLENDVPD** ymm1, ymm2, ymm3/m256, ymm4
- **VBLENDVPS** ymm1, ymm2, ymm3/m256, ymm4

4.1.9.5 256-bit Vector TEST Instructions

The 256-bit vector **TEST** instructions listed below are decomposed into two µops with dependence between them. The operation result is written in the GPR arithmetic flags. Throughput is one per cycle, and latency is seven cycles.

- **VTESTPS** ymm1, ymm2/m256
- **VTESTPD** ymm1, ymm2/m256
- **VPTEST** ymm1, ymm2/m256

4.1.9.6 The GATHER Instruction

The **VGATHER** instructions are implemented as micro-coded flow. Latency is ~50 cycles.

4.1.9.7 Masked Load and Store Instructions

The throughput of 256-bit VMASKMOV load and store is once every two cycles and that of the 128-bit VMASKMOV load and store is one per cycle.

A masked load or store with a masked element may encounter performance degradation if the masked element memory access causes an exception or a fault.

4.1.9.8 ADX Instructions

ADX instructions are supported. **ADCX** and **ADOX** are partial arithmetic flag-updating instructions. Intel Core microarchitecture renames and tracks arithmetic flags differently than Intel Atom. The carry flag (CF), overflow flag (OF), and other flags (ZF, AF, PF, SF) are renamed as independent registers on Core while they remain as a single register on Atom.

Unless there is a non-flag consuming full flag updating instruction between ADCX/ADOX instructions, on Crestmont, there is an operand dependency between the ADCX and ADOX instructions as the arithmetic flag register is a source operand of both. As this dependence between ADCX and ADOX instructions does not exist in the Intel Core microarchitecture, hand-tuned binaries exploiting this parallelism exist. While Crestmont supports the ISA, the parallelism will be lower on this microarchitecture.

4.2 THE TREMONT MICROARCHITECTURE

The Tremont microarchitecture builds on the success of the Goldmont Plus microarchitecture and provides the following enhancements:

- Enhanced branch prediction unit.
  - — Increased capacity with improved path-based conditional and indirect prediction.
  - — New committed Return Stack Buffer.
- Novel clustered 6-wide out-of-order front-end fetch and decode pipeline.
  - — Banked ICache with dual 16B reads.
  - — Two 3-wide decode clusters enabling up to 6 instructions per cycle.
- Deeper back-end out-of-order windows.
- 32KB data cache.
- Larger load and store buffers.

- Dual generic load and store execution pipes capable of two loads, two stores, or one load and store per cycle.
- Dedicated integer and vector integer/floating point store data ports.
- New and improved cryptography.
  - New Galois-field instructions (GFNI).
  - Dual AES units.
  - Enhanced SHA-NI implementation.
  - Faster PCLMULQDQ.
- Support for user-level low-power and low-latency spin-loop instructions UMWAIT/UMONITOR and TPAUSE.

4.2.1 TREMONT MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Tremont microarchitecture is depicted in Figure 4-4.

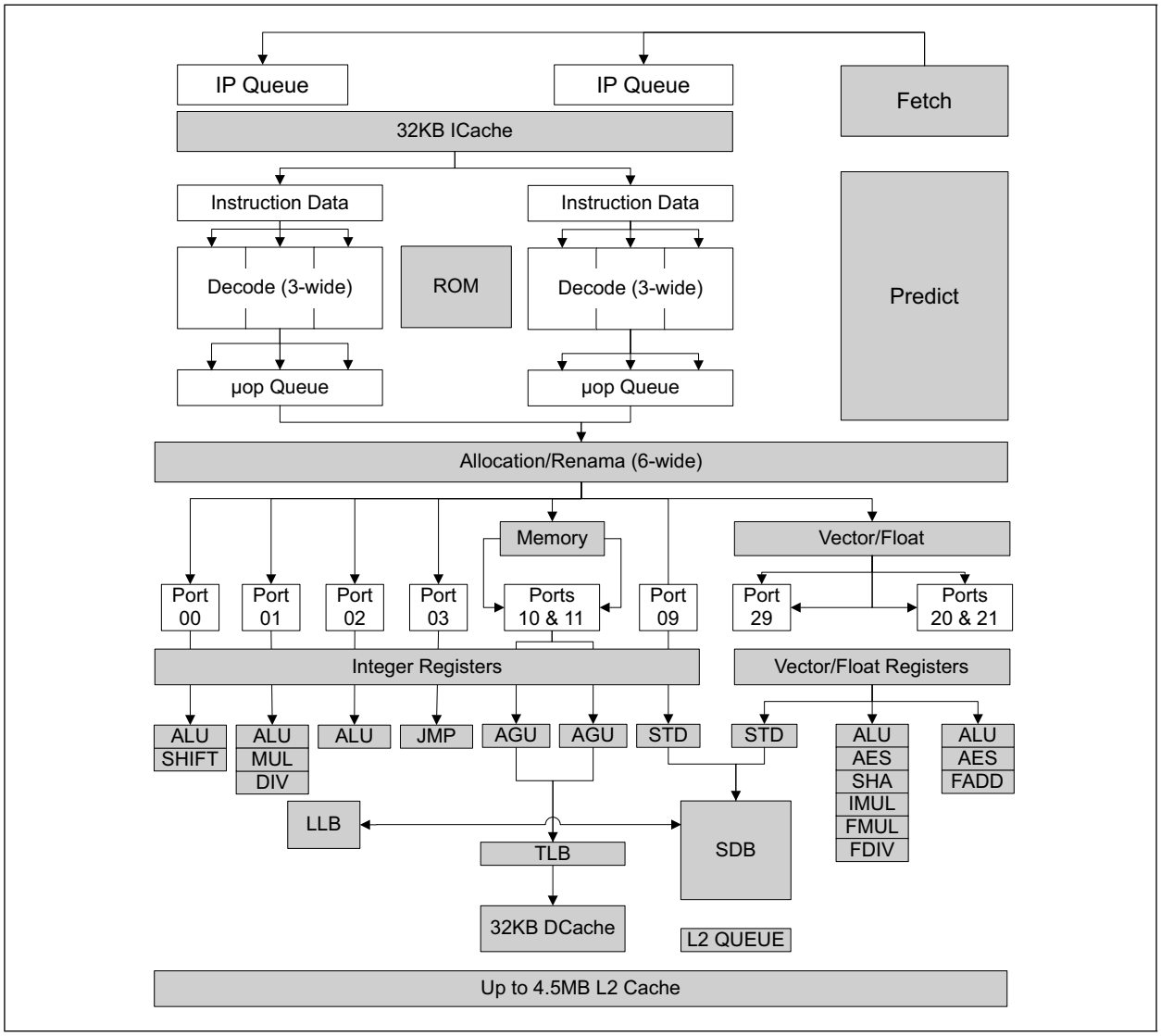


Figure 4-4. Processor Core Pipeline Functionality of the Tremont Microarchitecture

The Tremont microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of several components, including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

4.2.2 THE FRONT END

Tremont microarchitecture introduces parallel out-of-order instruction decode. Instruction pointers access the ITLB, check the ICache tag array, and access the branch predictor. When the branch predictor produces a taken branch target, the new block of code advances the decode cluster assignment.

Tremont microarchitecture has a 32B predict pipeline that feeds dual 3-wide decode clusters capable of 6 instruction decode per cycle. Each cluster can access a banked 32KB instruction cache at 16B/cycle for a maximum of 32B/cycle. Due to differences in the number of instructions per block and other decode latency differences, younger blocks of code can decode before older blocks. At the end of each decode cluster is a queue of decoded instructions (µop queue).

The allocation and rename pipeline reads both µop queues in parallel and puts the instruction stream back in order for register renaming and resource allocation. Whereas increasing decode width for x86 traditionally requires exponential resources and triggers efficiency loss, clustering allows for x86 decode to be built with linear resources and little efficiency loss.

As the clustering algorithm is dependent on the ability to predict taken branches within the branch predictor, very long assembly sequences that lack taken branches (long unrolled code utilizing the floating point unit, for example) can be bottlenecked due to being unable to utilize both decode clusters simultaneously. Inserting unconditional JMP instructions to the next sequential instruction pointer at intervals between 16 to 32 instructions may relieve this bottleneck if encountered.

While Tremont microarchitecture did not build a dynamic mechanism to load balance the decode clusters, future generations of Intel Atom processors will include hardware to recognize and mitigate these cases without the need for explicit insertions of taken branches into the assembly code.

In addition to the novel clustered decode scheme, Tremont microarchitecture enhanced the branch predictor and doubled the size of the L2 Predecode cache from 64KB on the Goldmont Plus microarchitecture to 128 KB.

The low level characteristics of the microarchitecture within each decode cluster remain the same as in the Goldmont Plus microarchitecture. For example, instructions should avoid more than 4 Bytes of prefixes and escapes.

4.2.3 THE OUT-OF-ORDER AND EXECUTION ENGINES

The Out of Order and execution engines changes in the Tremont microarchitecture include:

- A significant increase in size of
  - Reorder buffer.
  - Load buffer.
  - Store buffer.
  - Reservation stations which enable deeper OOO execution and higher cache bandwidth.
- **Wider machine:** 8 → 10 execution ports.
- Greater capabilities per execution port.

Table 4-2 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 4-2. Dispatch Port and Execution Stacks of the Tremont Microarchitecture

Port 00 INT	Port 01 INT	Port 02 INT	Port 08 INT	Port 09 INT	Port 10	Port 11	Port 20 FP/VEC	Port 21 FP/VEC	Port 29 FP/VEC
ALU LEA[1] Shift	ALU LEA[2] Bit Ops IMUL IDIV POPCNT CRC32	ALU LEA[3]	JUMP	Store Data	Load Store Address	Load Store Address	ALU AES SHA-RND FMUL FDIV Shuffle Shift SIMUL GFNI Converts	ALU AES SHA-MSG FADD Shuffle	Store Data

NOTES:

1. LEAs without a scaled index and only two sources (among base, index, and displacement inputs) execute as one operation on any ALU port (00, 01, or 02).
2. LEAs with three sources fracture into two operations and take an additional cycle of latency. Index consuming portion, regardless of scale value, will bind to port 02 while second operation binds to either port 00 or 01.
3. LEAs with a scaled index but without a displacement execute as one operation on port 02.

4.2.4 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy changes in Tremont microarchitecture include:

- 33% increase in size of the L1 data cache from 24KB to 32KB.
- 2×L1 load bandwidth:
  - 1 dedicated load port
  - 2 generic AGUs, shared between loads and stores.
- 2×L1 store bandwidth:
  - 1 dedicated store port
  - 2 generic AGUs, shared between loads and stores.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Maintains a 3-cycle load-to-use latency.
- Larger 2nd level TLB:
  - 512 4K entries → 1K 4K entries
  - 32 2M/4M entries → 64 2M/4M entries
- L2 cache size from 1MBs to 4.5MBs depending on SoC design choice:
  - The L2 size on Snow Ridge products is 4.5MBs, whereas the L2 size on Lakefield products is 1.5MBs.

The TLB hierarchy consists of a dedicated level one TLB for instruction cache and data cache with a shared second-level TLB for all page translations.

Table 4-3. Cache Parameters of the Tremont Microarchitecture

Level	Page Size	Entries	Associativity
Instruction	4KB/2M/4M[1]	48	Fully associative
First Level Data (loads and stores)	4KB/2M/4M[2]	32	Fully associative
Second Level	4KB	1024	4
Second Level	2M/4M	64	4

NOTES:

1. The first level instruction TLB (ITLB) caches small and large page translations but large pages are cached as 256KB regions per ITLB entry.
2. The first level data TLB (uTLB) caches small and large page translations but large pages are fully fractured into 4KB regions per uTLB entry.

4.2.5 NEW INSTRUCTIONS

New instructions and architectural changes in Tremont microarchitecture are listed below. Actual support may be product dependent.

- Galois Field New Instructions (GFNI) for
  - — Acceleration of various encryption algorithms,
  - — Error correction algorithms, and
  - — Bit matrix multiplications.
- UMWAIT/UMONITOR/TPAUSE instructions enable power savings in user level spin loops.
- Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Performance debugging benefits can be realized from the Tremont microarchitecture skidless PEBS implementation on both PMC0 and the fixed instruction counter. This enables a precise distribution via sampling on instructions and/or any of the precise general purpose events. As PEBS is triggered on the event **after** the overflow is signaled, counters should be programmed to large numbers that are (PRIME-1).

4.2.6 TREMONT MICROARCHITECTURE POWER MANAGEMENT

Tremont microarchitecture supports many of the same features as those found on the Ice Lake Client microarchitecture. Processors based on Tremont microarchitecture are the first Intel Atom processors with support for Intel® Speed Shift Technology. Power management features sometimes differ depending on the needs of the SoC.

CHAPTER 5
CODING FOR SIMD ARCHITECTURES

- Processors based on Intel Core microarchitecture support MMX™, Intel® SSE, Intel® SSE2, Intel® SSE3, and Intel® SSSE3.
- Processors based on Enhanced Intel Core microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.
- Processors based Westmere microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, Intel SSE4.1, Intel SSE4.2, and AESNI.
- Processors based on Sandy Bridge microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, Intel SSE4.1, Intel SSE4.2, AESNI, PCLMULQDQ, and Intel® AVX.
- Intel® Core™ Solo and Intel® Core™ Duo processors support MMX, Intel SSE, Intel SSE2, and Intel SSE3.

Single-instruction, multiple-data (SIMD) technologies enable the development of advanced multimedia, signal processing, and modeling applications.

SIMD techniques can be applied to text/string processing, lexing and parser applications. This is covered in Chapter 14, "Intel® SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing." Techniques for optimizing AESNI are discussed in Section 6.10.

To take advantage of the performance opportunities presented by these capabilities, do the following:

- Ensure that the processor supports MMX technology, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.
- Ensure that the operating system supports MMX technology and Intel SSE (OS support for Intel SSE2, Intel SSE3 and Intel SSSE3 is the same as OS support for Intel SSE).
- Employ the optimization and scheduling strategies described in this book.
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Utilize the cacheability instructions offered by Intel SSE and Intel SSE2, where appropriate.

5.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES

This section shows how to check whether a processor supports MMX technology, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.

SIMD technology can be included in your application in three ways:

1. Check for the SIMD technology during installation. If the desired SIMD technology is available, the appropriate DLLs can be installed.
2. Check for the SIMD technology during program execution and install the proper DLLs at runtime. This is effective for programs that may be executed on different machines.
3. Create a "fat" binary that includes multiple versions of routines; versions that use SIMD technology and versions that do not. Check for SIMD technology during program execution and run the appropriate versions of the routines. This is especially effective for programs that may be executed on different machines.

5.1.1 CHECKING FOR MMX TECHNOLOGY SUPPORT

If MMX technology is available, then CPUID.01H:EDX[BIT 23] = 1. Use the code segment in Example 5-1 to test for MMX technology.

Example 5-1. Identification of MMX Technology with CPUID

```
…identify existence of cpuid instruction
…                           ;
…                           ; Identify signature is genuine Intel
…                           ;
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H CPUID instruction
test edx, 00800000h         ; Is MMX technology bit (bit 23) in feature flags equal to 1
jnz       Found
```

See CPUID Information for Intel® Processors for more information.

5.1.2 CHECKING FOR INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) SUPPORT

Checking for processor support of Intel Streaming SIMD Extensions (SIntel SE) on your processor is similar to checking for MMX technology. However, operating system (OS) must provide support for Intel SSE states save and restore on context switches to ensure consistent application behavior when using Intel SSE instructions.

To check whether your system supports Intel SSE, follow these steps:

1. Check that your processor supports the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSE existence.

Example 5-2 shows how to find the SSE feature bit (bit 25) in CPUID feature flags.

Example 5-2. Identification of Intel® SSE with CPUID

```
…Identify existence of cpuid instruction
                            ; Identify signature is genuine intel
mov eax, 1                  ; Request for feature flags
cpuid                       ; 0FH, 0A2H cpuid instruction
test EDX, 002000000h        ; Bit 25 in feature flags equal to 1
jnz             Found
```

5.1.3 CHECKING FOR INTEL® STREAMING SIMD EXTENSIONS 2 (INTEL® SSE2) SUPPORT

Checking for support of Intel SSE2 is like checking for Intel SSE support. The OS requirements for Intel SSE2 Support are the same as the OS requirements for Intel SSE.

To check whether your system supports Intel SSE2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSE2 technology existence.

Example 5-3 shows how to find the SSE2 feature bit (bit 26) in the CPUID feature flags.

Example 5-3. Identification of Intel® SSE2 with cpuid

```
    …identify existence of cpuid instruction
    …                           ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test EDX, 004000000h            ; Bit 26 in feature flags equal to 1
jnz       Found
```

5.1.4 CHECKING FOR INTEL® STREAMING SIMD EXTENSIONS 3 (INTEL® SSE3) SUPPORT

Intel SSE3 includes 13 instructions, 11 of those are suited for SIMD or x87 style programming. Checking for support of Intel SSE3 instructions is similar to checking for Intel SSE support. The OS requirements for Intel SSE3 Support are the same as the requirements for Intel SSE.

To check whether your system supports the x87 and SIMD instructions of Intel SSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the ECX feature bit 0 of CPUID for Intel SSE3 technology existence.

Example 5-4 shows how to find the SSE3 feature bit (bit 0 of ECX) in the CPUID feature flags.

Example 5-4. Identification of Intel® SSE3 with CPUID

```
    …identify existence of cpuid instruction
    …                           ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000000001h            ; Bit 0 in feature flags equal to 1
jnz       Found
```

Software must check for support of MONITOR and MWAIT before attempting to use MONITOR and MWAIT.Detecting the availability of MONITOR and MWAIT can be done using a code sequence similar to Example 5-4. The availability of MONITOR and MWAIT is indicated by bit 3 of the returned value in ECX.

5.1.5 CHECKING FOR INTEL® SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (INTEL® SSSE) SUPPORT

Checking for support of Intel SSSE3 is similar to checking for Intel SSE support. The OS requirements for Intel SSSE3 support are the same as the requirements for Intel SSE.

To check whether your system supports Intel SSSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSSE3 technology existence.

Example 5-5 shows how to find the Intel SSSE3 feature bit in the CPUID feature flags.

Example 5-5. Identification of SSSE3 with cpuid

```
    …Identify existence of CPUID instruction
    …                          ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test ECX, 000000200h           ; ECX bit 9
jnz      Found
```

5.1.6 CHECKING FOR INTEL® SSE4.1 SUPPORT

Checking for support of SSE4.1 is similar to checking for Intel SSE support. The OS requirements for Intel SSE4.1 support are the same as the requirements for Intel SSE.

To check whether your system supports Intel SSE4.1, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for Intel SSE4.1.

Example 5-6 shows how to find the Intel SSE4.1 feature bit in the CPUID feature flags.

Example 5-6. Identification of Intel® SSE4.1 with CPUID

```
    …Identify existence of CPUID instruction
    …                          ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test ECX, 000080000h           ; ECX bit 19
jnz      Found
```

5.1.7 CHECKING FOR INTEL® SSE4.2 SUPPORT

Checking for support of Intel SSE4.2 is similar to checking for Intel SSE support. The OS requirements for SSE4.2 support are the same as the requirements for Intel SSE.

To check whether your system supports SSE4.2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for Intel SSE4.2.

Example 5-7 shows how to find the INtel SSE4.2 feature bit in the CPUID feature flags.

Example 5-7. Identification of SSE4.2 with cpuid

```
    …Identify existence of CPUID instruction
    …                          ; Identify signature is genuine intel
mov eax, 1                     ; Request for feature flags
cpuid                          ; 0FH, 0A2H CPUID instruction
test ECX, 000100000h           ; ECX bit 20
jnz      Found
```

5.1.8 DETECTION OF PCLMULQDQ AND AESNI INSTRUCTIONS

Before an application attempts to use the following AESNI instructions: AESDEC/AESDECLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AESNI extensions. AESNI extensions is supported if CPUID.01H:ECX.AESNI[bit 25] = 1.

Prior to using PCLMULQDQ instruction, application must check if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling SSE state will also support applications that use AESNI extensions and PCLMULQDQ instruction. This is the same requirement for Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.

Example 5-8. Detection of AESNI Instructions

```
    …Identify existence of CPUID instruction
    …                           ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 002000000h            ; ECX bit 25
jnz      Found
```

Example 5-9. Detection of PCLMULQDQ Instruction

```
    …Identify existence of CPUID instruction
    …                           ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000000002h            ; ECX bit 1
jnz      Found
```

5.1.9 DETECTION OF INTEL® AVX INSTRUCTIONS

Intel AVX operates on the 256-bit YMM register state. Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 5-1.

Prior to using AVX, the application must identify that the operating system supports the XGETBV instruction, the YMM register state, in addition to processor's support for YMM state management using XSAVE/XRSTOR and AVX instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use[1])

2) Issue XGETBV and verify that XFEATURE_ENABLED_MASK[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Detect CPUID.1:ECX.AVX[bit 28] = 1 (AVX instructions supported). This can be done in any order relative to 1 and 2.

1.If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XFEATURE_ENALBED_MASK register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

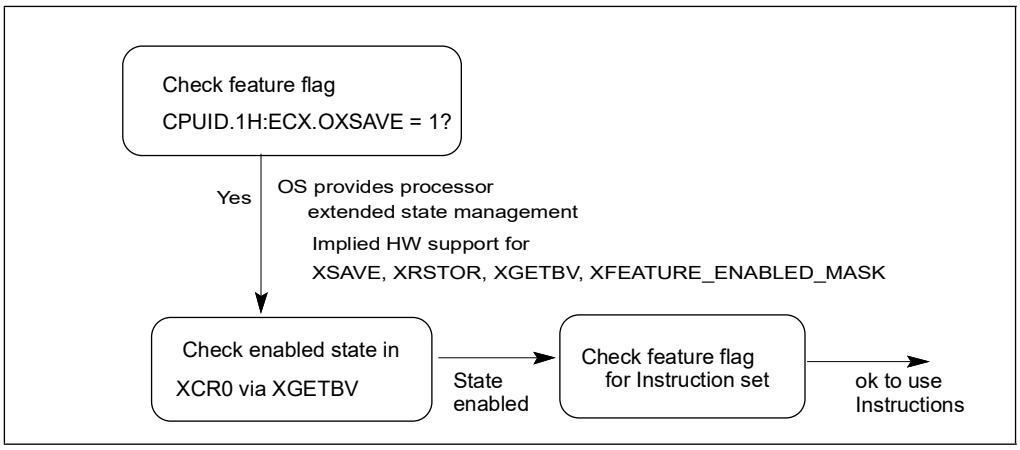


Figure 5-1. General Procedural Flow of Application Detection of Intel® AVX

The following pseudocode illustrates this recommended application Intel AVX detection process:

Example 5-10. Detection of Intel® AVX Instruction

```
INT supports_AVX()
{    mov       eax, 1
     cpuid
     and       ecx, 018000000H
     cmp       ecx, 018000000H; check both OSXSAVE and AVX feature flags
      jne      not_supported
     ; processor supports AVX instructions and XGETBV is enabled by OS
     mov       ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV              ; result in EDX:EAX
     and       eax, 06H
     cmp       eax, 06H; check OS has enabled both XMM and YMM state support
     jne       not_supported
     mov       eax, 1
     jmp       done
NOT_SUPPORTED:
     mov       eax, 0
done:
```

NOTE

It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating systems, AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. “CPUID.1:ECX.XSAVE[bit 26] = 1” does not guarantee the OS actually uses the XSAVE process for state management.

5.1.10 DETECTION OF VEX-ENCODED AES AND VPCLMULQDQ

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for Intel AVX.

Example 5-11. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VAESNI()
{   mov     eax, 1
    cpuid
    and     ecx, 01A000000H
    cmp     ecx, 01A000000H; check OSXSAVE AVX and AESNI feature flags
     jne    not_supported
    ; processor supports AVX and VEX-encoded AESNI and XGETBV is enabled by OS
    mov     ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
    XGETBV          ; result in EDX:EAX
    and     eax, 06H
    cmp     eax, 06H; check OS has enabled both XMM and YMM state support
    jne     not_supported
    mov     eax, 1
    jmp     done
NOT_SUPPORTED:
    mov     eax, 0
done:
```

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 and the sequence for detection application support for AVX.

This is shown in the pseudocode:

Example 5-12. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VPCLMULQDQ)
{   mov     eax, 1
    cpuid

    and     ecx, 018000002H
    cmp     ecx, 018000002H; check OSXSAVE AVX and PCLMULQDQ feature flags
     jne    not_supported
    ; processor supports AVX and VEX-encoded PCLMULQDQ and XGETBV is enabled by OS
    mov     ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
    XGETBV          ; result in EDX:EAX
    and     eax, 06H
    cmp     eax, 06H; check OS has enabled both XMM and YMM state support
    jne     not_supported
    mov     eax, 1
    jmp     done
NOT_SUPPORTED:
    mov     eax, 0
done:
```

5.1.11 DETECTION OF F16C INSTRUCTIONS

Application using float 16 instruction must follow a detection sequence similar to Intel AVX to ensure:

- The OS has enabled YMM state management support.
- The processor support Intel AVX as indicated by the CPUID feature flag, i.e. CPUID.01H:ECX.AVX[bit 28] = 1.
- The processor support 16-bit floating-point conversion instructions via a CPUID feature flag (CPUID.01H:ECX.F16C[bit 29] = 1).

Application detection of Float-16 conversion instructions follow the general procedural flow in Figure 5-2.

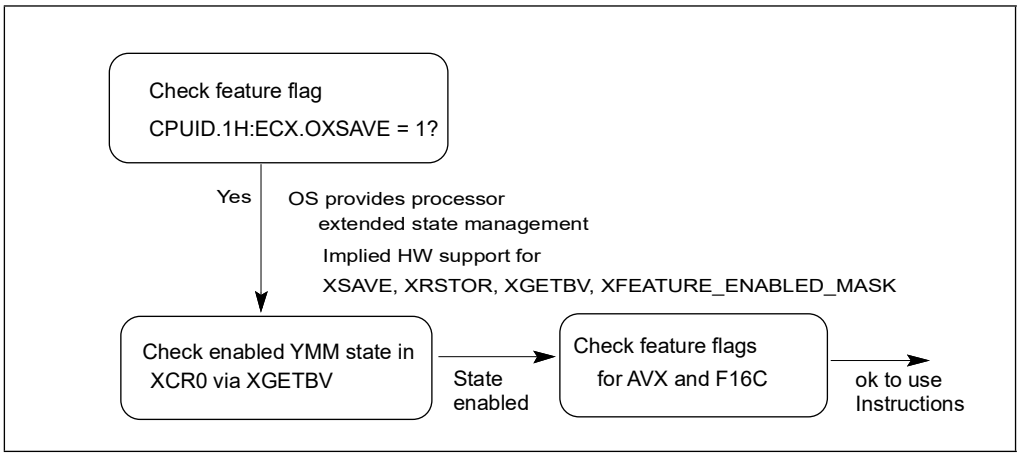


Figure 5-2. General Procedural Flow of Application Detection of Float-16

```
INT supports_f16c()
{                       ; result in eax
                        mov eax, 1
                        cpuid
                        and ecx, 038000000H
                        cmp ecx, 038000000H; check OSXSAVE, AVX, F16C feature flags
                        jne not_supported
                        ; processor supports AVX,F16C instructions and XGETBV is enabled by OS
                        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
                        XGETBV; result in EDX:EAX
                        and eax, 06H
                        cmp eax, 06H; check OS has enabled both XMM and YMM state support
                        jne not_supported
                        mov eax, 1
                        jmp done
                        NOT_SUPPORTED:
```

```
                    mov eax, 0
                    done:
}
```

5.1.12 DETECTION OF FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:

```
INT supports_fma()
{                   ; result in eax
                    mov eax, 1
                    cpuid
                    and ecx, 018001000H
                    cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
                     jne not_supported
                    ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
                    mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
                    XGETBV; result in EDX:EAX
                    and eax, 06H
                    cmp eax, 06H; check OS has enabled both XMM and YMM state support
                    jne not_supported
                    mov eax, 1
                    jmp done
                    NOT_SUPPORTED:
                    mov eax, 0
                    done:
}
```

5.1.13 DETECTION OF INTEL® AVX2

Hardware support for Intel AVX2 is indicated by CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1.

Application Software must identify that hardware supports Intel AVX, after that it must also detect support for AVX2 by checking CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]. The recommended pseudocode sequence for detection of Intel AVX2 is:

```
INT supports_avx2()
{                       ; result in eax
                        mov eax, 1
                        cpuid
                        and ecx, 018000000H
                        cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
                        jne not_supported
                        ; processor supports AVX instructions and XGETBV is enabled by OS
                        mov eax, 7
                        mov ecx, 0
                        cpuid
                        and ebx, 20H
                        cmp ebx, 20H; check AVX2 feature flags
                        jne not_supported
                        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
                        XGETBV; result in EDX:EAX
                        and eax, 06H
                        cmp eax, 06H; check OS has enabled both XMM and YMM state support
                        jne not_supported
                        mov eax, 1
                        jmp done
                        NOT_SUPPORTED:
                        mov eax, 0
                        done:
}
```

5.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING

The VTune Performance Enhancement Environment CD provides tools to aid in the evaluation and tuning. Before implementing them, you need answers to the following questions:

1. Will the current code benefit by using MMX technology, Intel SSE, Intel SSE2, Intel SSE3, or Intel SSSE3?
2. Is this code integer or floating-point?
3. What integer word size or floating-point precision is needed?
4. What coding techniques should I use?
5. What guidelines do I need to follow?
6. How should I arrange and align the datatypes?

Figure 5-3 provides a flowchart for the process of converting code to MMX technology, Intel SSE, Intel SSE2, Intel SSE3, or Intel SSSE3.

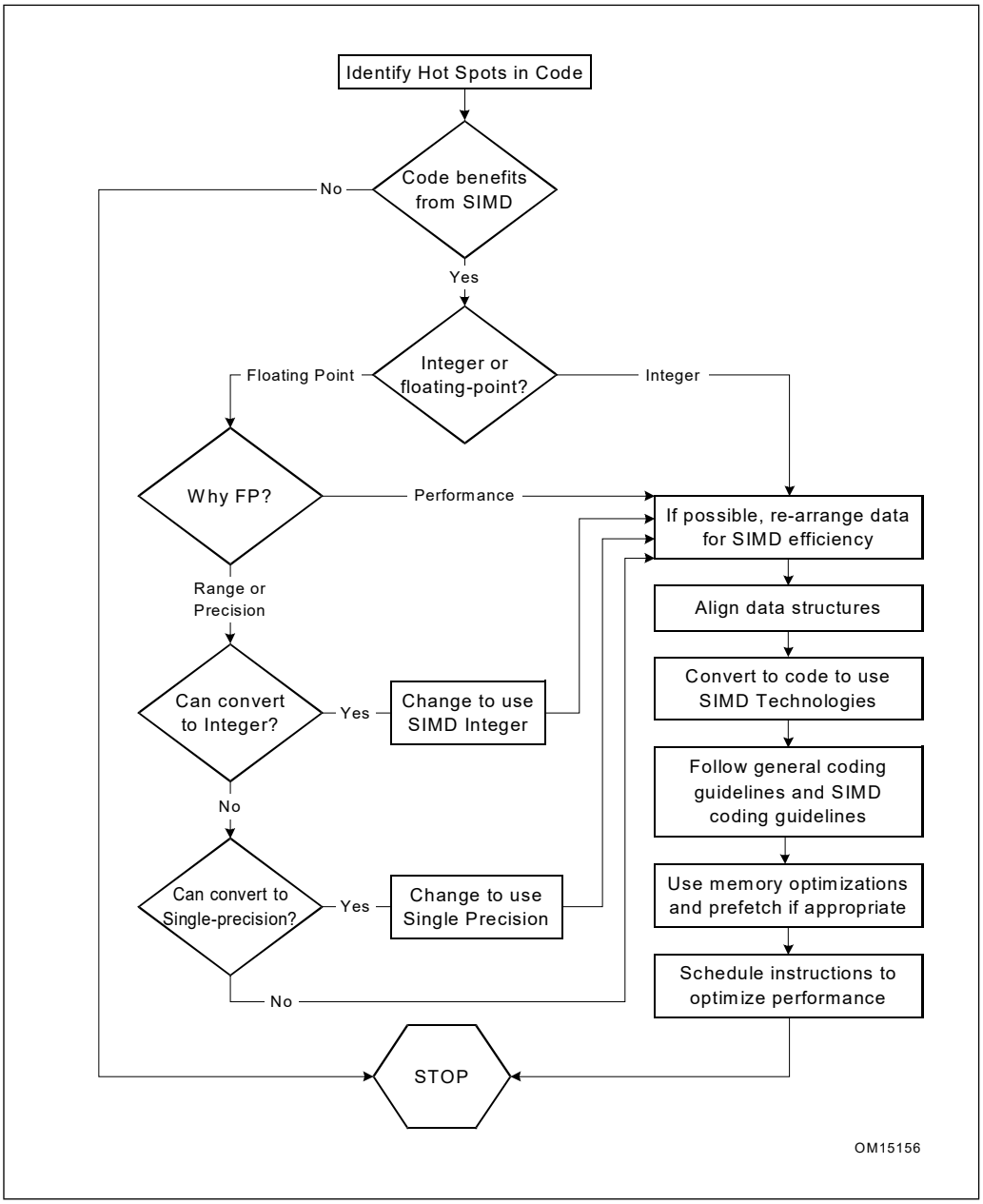


Figure 5-3. Converting to Intel® Streaming SIMD Extensions Chart

To use any of the SIMD technologies optimally, you must evaluate the following situations in your code:

- Fragments that are computationally intensive.
- Fragments that are executed often enough to have an impact on performance.
- Fragments that with little data-dependent control flow.
- Fragments that require floating-point computations.
- Fragments that can benefit from moving data 16 bytes at a time.
- Fragments of computation that can coded using fewer instructions.
- Fragments that require help in using the cache hierarchy efficiently.

5.2.1 IDENTIFYING HOT SPOTS

To optimize performance, use the VTune Performance Analyzer to find sections of code that occupy most of the computation time. Such sections are called the hotspots. See Appendix A, "Application Performance Tools."

The VTune analyzer provides a hotspots view of a specific module to help you identify sections in your code that take the most CPU time and that have potential performance problems. The hotspots view helps you identify sections in your code that take the most CPU time and that have potential performance problems.

The VTune analyzer enables you to change the view to show hotspots by memory location, functions, classes, or source files. You can double-click on a hotspot and open the source or assembly view for the hotspot and see more detailed information about the performance of each instruction in the hotspot.

The VTune analyzer offers focused analysis and performance data at all levels of your source code and can also provide advice at the assembly language level. The code coach analyzes and identifies opportunities for better performance of C/C++, Fortran and Java* programs, and suggests specific optimizations. Where appropriate, the coach displays pseudo-code to suggest the use of highly optimized intrinsics and functions in the Intel® Performance Library Suite. Because VTune analyzer is designed specifically for Intel architecture (IA)-based processors, it can offer detailed approaches to working with IA. See Appendix A.1.1 for details.

5.2.2 DETERMINE IF CODE BENEFITS BY CONVERSION TO SIMD EXECUTION

Identifying code that benefits by using SIMD technologies can be time-consuming and difficult. Likely candidates for conversion are applications that are highly computation intensive, such as the following:

- Speech compression algorithms and filters.
- Speech recognition algorithms.
- Video display and capture routines.
- Rendering routines.
- 3D graphics (geometry).
- Image and video processing algorithms.
- Spatial (3D) audio.
- Physical modeling (graphics, CAD).
- Workstation applications.
- Encryption algorithms.
- Complex arithmetics.

Generally, good candidate code is code that contains small-sized repetitive loops that operate on sequential arrays of integers of 8, 16 or 32 bits, single-precision 32-bit floating-point data, double precision 64-bit floating-point data (integer and floating-point data items should be sequential in memory). The repetitiveness of these loops incurs costly application processing time. However, these routines have potential for increased performance when you convert them to use one of the SIMD technologies.

Once you identify your opportunities for using a SIMD technology, you must evaluate what should be done to determine whether the current algorithm or a modified one will ensure the best performance.

5.3 CODING TECHNIQUES

The SIMD features of Intel SSE3, Intel SSE2, Intel SSE, and MMX technology require new methods of coding algorithms. One of them is vectorization. Vectorization is the process of transforming sequentially-executing, or

scalar, code into code that can execute in parallel, taking advantage of the SIMD architecture parallelism. This section discusses the coding techniques available for an application to make use of the SIMD architecture.

To vectorize your code and thus take advantage of the SIMD architecture, do the following:

- Determine if the memory accesses have dependencies that would prevent parallel execution.
- "Strip-mine" the inner loop to reduce the iteration count by the length of the SIMD operations (for example, four for single-precision floating-point SIMD, eight for 16-bit integer SIMD on the XMM registers).
- Re-code the loop with the SIMD instructions.

Each of these actions is discussed in detail in the subsequent sections of this chapter. These sections also discuss enabling automatic vectorization using the Intel C++ Compiler.

5.3.1 CODING METHODOLOGIES

Software developers need to compare the performance improvement that can be obtained from assembly code versus the cost of those improvements. Programming directly in assembly language for a target platform may produce the required performance gain, however, assembly code is not portable between processor architectures and is expensive to write and maintain.

Performance objectives can be met by taking advantage of the different SIMD technologies using high-level languages as well as assembly. The new C/C++ language extensions designed specifically for Intel SSE3, Intel SSE2, Intel SSE, and MMX technology help make this possible.

Figure 5-4 illustrates the trade-offs involved in the performance of hand-coded assembly versus the ease of programming and portability.

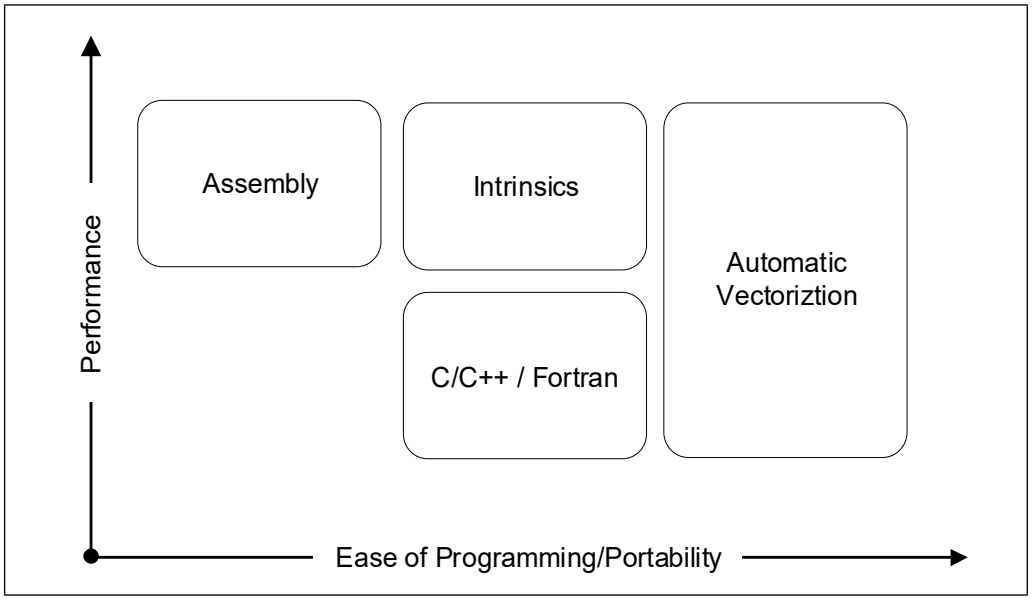


Figure 5-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-Offs

The examples that follow illustrate the use of coding adjustments to enable the algorithm to benefit from the Intel SSE. The same techniques may be used for single-precision floating-point, double-precision floating-point, and integer data under Intel SSE3, Intel SSE2, Intel SSE, and MMX technology.

As a basis for the usage model discussed in this section, consider a simple loop shown in Example 5-13.

Example 5-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
   c[i] = a[i] + b[i];
 }
}
```

Note that the loop runs for only four iterations. This allows a simple replacement of the code with Streaming SIMD Extensions.

For the optimal use of the Intel SSE that need data alignment on the 16-byte boundary, all examples in this chapter assume that the arrays passed to the routine, *A*, *B*, *C*, are aligned to 16-byte boundaries by a calling routine.

The sections that follow provide details on the coding methodologies: inlined assembly, intrinsics, C++ vector classes, and automatic vectorization.

5.3.1.1 Assembly

Key loops can be coded directly in assembly language using an assembler or by using inlined assembly (C-asm) in C/C++ code. The Intel compiler or assembler recognize the new instructions and registers, then directly generate the corresponding code. This model offers the opportunity for attaining greatest performance, but this performance is not portable across the different processor architectures.

Example 5-14 shows the Intel SSE inlined assembly encoding.

Example 5-14. Intel® Streaming SIMD Extensions (Intel® SSE) Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
 __asm {
   mov    eax, a
   mov    edx, b
   mov    ecx, c
   movaps  xmm0, XMMWORD PTR [eax]
   addps   xmm0, XMMWORD PTR [edx]
   movaps  XMMWORD PTR [ecx], xmm0
 }
}
```

5.3.1.2 Intrinsics

Intrinsics provide the access to the ISA functionality using C/C++ style coding instead of assembly language. Intel has defined three sets of intrinsic functions that are implemented in the Intel C++ Compiler to support the MMX technology, Intel SSE, Intel SSE2. Four new C data types, representing 64-bit and 128-bit objects are used as the operands of these intrinsic functions. __M64 is used for MMX integer SIMD, __M128 is used for single-precision floating-point SIMD, __M128I is used for Streaming SIMD Extensions 2 integer SIMD, and __M128D is used for

double precision floating-point SIMD. These types enable the programmer to choose the implementation of an algorithm directly, while allowing the compiler to perform register allocation and instruction scheduling where possible. The intrinsics are portable among all Intel architecture-based processors supported by a compiler.

The use of intrinsics allows you to obtain performance close to the levels achievable with assembly. The cost of writing and maintaining programs with intrinsics is considerably less. For a detailed description of the intrinsics and their use, refer to the Intel C++ Compiler documentation.

Example 5-15 shows the loop from Example 5-13 using intrinsics.

Example 5-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

The intrinsics map one-to-one with actual Intel SSE assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an mm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, "+", ">>").
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the Intel Architecture MMX Technology Developer's Guide. For a description of data types, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

5.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Intel SSE and Intel SSE2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance of applications using this methodology can approach that of one using the intrinsics. Further details on the use of these classes can be found in the Intel C++ Class Libraries for SIMD Operations page.

Example 5-16 shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries.

Example 5-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Here, fvec.h is the class definition file and F32vec4 is the class representing an array of four floats. The "+" and "=" operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundary.

5.3.1.4 Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which loops, such as in Example 5-13 can be automatically vectorized, or converted into Intel SSE code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

- The layout of the loop and the data structures used.
- Dependencies amongst the data accesses in each iteration and across iterations.

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

Example 5-17 shows the code for automatic vectorization for the simple four-iteration loop (from Example 5-13).

Example 5-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
                float *restrict b,
                float *restrict c)
{
        int i;
        for (i = 0; i < 4; i++) {
                c[i] = a[i] + b[i];
        }
}
```

Compile this code using the -QAX and -QRESTRICT switches of the Intel C++ Compiler, version 4.0 or later.

The RESTRICT qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used, provides the only means of accessing the memory in question in the scope in which the pointers live. Without the restrict qualifier, the compiler will still vectorize this loop using runtime data dependence testing, where the generated code dynamically selects between sequential or vector execution of the loop, based on overlap of the parameters. The restrict keyword avoids the associated overhead altogether.

See Intel® C++ Compiler Classic Developer Guide and Reference for details.

5.4 STACK AND DATA ALIGNMENT

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

5.4.1 ALIGNMENT AND CONTIGUITY OF DATA ACCESS PATTERNS

The 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Intel SSE and Intel SSE2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and SSE. Several techniques for improving data access, such as padding, organizing data elements into arrays, etc. are described below. Intel SSE3 provides a special-purpose instruction LDDQU that can avoid cache line splits is discussed in Section 6.7.3

5.4.1.1 Using Padding to Align Data

However, when accessing SIMD data using SIMD operations, access to data can be improved simply by a change in the declaration. For example, consider a declaration of a structure, which represents a point in space plus an attribute.

```
typedef struct {short x,y,z; char a} Point;
Point pt[N];
```

Assume we will be performing several computations on X, Y, Z in three of the four elements of a SIMD word; see Section 5.5.1 for an example. Even if the first element in array PT is aligned, the second element will start 7 bytes later and not be aligned (3 shorts at two bytes each plus a single byte = 7 bytes).

By adding the padding variable PAD, the structure is now 8 bytes, and if the first element is aligned to 8 bytes (64 bits), all following elements will also be aligned. The sample declaration follows:

```
typedef struct {short x,y,z; char a; char pad;} Point;
Point pt[N];
```

5.4.1.2 Using Arrays to Make Data Contiguous

In the following code,

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

the second dimension Y needs to be multiplied by a scaling value. Here, the FOR loop accesses each Y dimension in the array PT thus disallowing the access to contiguous data. This can degrade the performance of the application by increasing cache misses, by poor utilization of each cache line that is fetched, and by increasing the chance for accesses which span multiple cache lines.

The following declaration allows you to vectorize the scaling operation and further improve the alignment of the data access patterns:

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty[i] *= scale;
```

With the SIMD technology, choice of data organization becomes more important and should be made carefully based on the operations that will be performed on the data. In some applications, traditional data arrangements may not lead to the maximum performance.

A simple example of this is an FIR filter. An FIR filter is effectively a vector dot product in the length of the number of coefficient taps.

Consider the following code:

(data [j] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]),

If in the code above the filter operation of data element I is the vector dot product that begins at data element J, then the filter operation of data element I+1 begins at data element J+1.

Assuming you have a 64-bit aligned data vector and a 64-bit aligned coefficients vector, the filter operation on the first data element will be fully aligned. For the second data element, however, access to the data vector will be misaligned. For an example of how to avoid the misalignment problem in the FIR filter, refer to Intel application notes on Streaming SIMD Extensions and filters.

Duplication and padding of data structures can be used to avoid the problem of data accesses in algorithms which are inherently misaligned. Section 5.5.1 discusses trade-offs for organizing data structures.

NOTE

The duplication and padding technique overcomes the misalignment problem, thus avoiding the expensive penalty for misaligned data access, at the cost of increasing the data size. When developing your code, you should consider this tradeoff and use the option which gives the best performance.

5.4.2 STACK ALIGNMENT FOR 128-BIT SIMD TECHNOLOGIES

For best performance, the Streaming SIMD Extensions and Streaming SIMD Extensions 2 require their memory operands to be aligned to 16-byte boundaries. Unaligned data can cause significant performance penalties compared to aligned data. However, the existing software conventions for IA-32 (STDCALL, CDECL, FASTCALL) as implemented in most compilers, do not provide any mechanism for ensuring that certain local data and certain parameters are 16-byte aligned. Therefore, Intel has defined a new set of IA-32 software conventions for alignment to support the new __M128* datatypes (__M128, __M128D, and __M218I). These meet the following conditions:

- Functions that use Streaming SIMD Extensions or Streaming SIMD Extensions 2 data need to provide a 16-byte aligned stack frame.
- __M128* parameters need to be aligned to 16-byte boundaries, possibly creating “holes” (due to padding) in the argument block.

The new conventions presented in this section as implemented by the Intel C++ Compiler can be used as a guideline for an assembly language code as well. In many cases, this section assumes the use of the __M128* data types, as defined by the Intel C++ Compiler, which represents an array of four 32-bit floats.

5.4.3 DATA ALIGNMENT FOR MMX™ TECHNOLOGY

Many compilers enable alignment of variables using controls. This aligns variable bit lengths to the appropriate boundaries. If some of the variables are not appropriately aligned as specified, you can align them using the C algorithm in Example 5-18.

Example 5-18. C Algorithm for 64-bit Data Alignment

```
/* Make newp a pointer to a 64-bit aligned array of NUM_ELEMENTS 64-bit elements. */
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```

The algorithm in Example 5-18 aligns an array of 64-bit elements on a 64-bit boundary. The constant of 7 is derived from one less than the number of bytes in a 64-bit element, or 8-1. Aligning data in this manner avoids the significant performance penalties that can occur when an access crosses a cache line boundary.

Another way to improve data alignment is to copy the data into locations that are aligned on 64-bit boundaries. When the data is accessed frequently, this can provide a significant performance improvement.

5.4.4 DATA ALIGNMENT FOR 128-BIT DATA

Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers used by Intel SSE, Intel SSE2, Intel SSE3, and Intel SSSE3. This must be done to avoid severe performance penalties and, at worst, execution faults.

There are MOVE instructions (and intrinsics) that allow unaligned data to be copied to and out of XMM registers when not using aligned data, but such operations are much slower than aligned accesses. If data is not 16-byte-aligned and the programmer or the compiler does not detect this and uses the aligned instructions, a fault occurs. So keep data 16-byte-aligned. Such alignment also works for MMX technology code, even though MMX technology only requires 8-byte alignment.

5.4.4.1 Compiler-Supported Alignment

The Intel C++ Compiler provides the following methods to ensure that the data is aligned.

Alignment by F32vec4 or __m128 Data Types

When the compiler detects F32VEC4 or __M128 data declarations or parameters, it forces alignment of the object to a 16-byte boundary for both global and local data, as well as parameters. If the declaration is within a function, the compiler also aligns the function's stack frame to ensure that local data and parameters are 16-byte-aligned. For details on the stack frame layout that the compiler generates for both debug and optimized (“release”-mode) compilations, refer to Intel’s compiler documentation.

__declspec(align(16)) specifications

These can be placed before data declarations to force 16-byte alignment. This is useful for local or global data declarations that are assigned to 128-bit data types. The syntax for it is

__declspec(align(integer-constant))

where the *INTEGER-CONSTANT* is an integral power of two but no greater than 32. For example, the following increases the alignment to 16-bytes:

__declspec(align(16)) float buffer[400];

The variable BUFFER could then be used as if it contained 100 objects of type __M128 or F32VEC4. In the code below, the construction of the F32VEC4 object, X, will occur with aligned data.

```
void foo() {
    F32vec4 x = *(__m128 *) buffer;
    ...
}
```

Without the declaration of __DECLSPEC(ALIGN(16)), a fault may occur.

Alignment by Using a UNION Structure

When feasible, a UNION can be used with 128-bit data types to allow the compiler to align the data structure by default. This is preferred to forcing alignment with __DECLSPEC(ALIGN(16)) because it exposes the true program intent to the compiler in that __M128 data is being used. For example:

```
union {
    float f[400];
    __m128 m[100];
} buffer;
```

Now, 16-byte alignment is used by default due to the __M128 type in the UNION; it is not necessary to use __DECLSPEC(ALIGN(16)) to force the result.

In C++ (but not in C) it is also possible to force the alignment of a CLASS/STRUCT/UNION type, as in the code that follows:

```
struct __declspec(align(16)) my_m128
{
     float f[4];
};
```

If the data in such a CLASS is going to be used with the Intel SSE or Intel SSE2, it is preferable to use a UNION to make this explicit. In C++, an anonymous UNION can be used to make this more convenient:

```
class my_m128 {
    union {
        __m128 m;
        float f[4];
    };
};
```

Because the UNION is anonymous, the names, M and F, can be used as immediate member names of MY__M128. Note that __DECLSPEC(ALIGN) has no effect when applied to a CLASS, STRUCT, or UNION member in either C or C++.

Alignment by Using __m64 or DOUBLE Data

In some cases, the compiler aligns routines with __M64 or DOUBLE data to 16-bytes by default. The command-line switch, -QSFALIGN16, limits the compiler so that it only performs this alignment on routines that contain 128-bit data. The default behavior is to use -QSFALIGN8. This switch instructs the complier to align routines with 8- or 16-byte data types to 16 bytes.

See Intel® C++ Compiler Classic Developer Guide and Reference for details.

5.5 IMPROVING MEMORY UTILIZATION

Memory performance can be improved by rearranging data and algorithms for Intel SSE, Intel SSE2, and MMX technology intrinsics. Methods for improving memory performance involve working with the following:

- Data structure layout.
- Strip-mining for vectorization and memory utilization.
- Loop-blocking.

Using the cacheability instructions, prefetch and streaming store, also greatly enhance memory utilization. See also: Chapter 9, "Optimizing Cache Usage."

5.5.1 DATA STRUCTURE LAYOUT

For certain algorithms, like 3D transformations and lighting, there are two basic ways to arrange vertex data. The traditional method is the array of structures (AoS) arrangement, with a structure for each vertex (Example 5-19). However this method does not take full advantage of SIMD technology capabilities.

Example 5-19. AoS Data Structure

```
typedef struct{
    float x,y,z;
    int a,b,c;
    . . .
} Vertex;
Vertex Vertices[NumOfVertices];
```

The best processing method for code using SIMD technology is to arrange the data in an array for each coordinate (Example 5-20). This data arrangement is called structure of arrays (SoA).

Example 5-20. SoA Data Structure

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int a[NumOfVertices];
    int b[NumOfVertices];
    int c[NumOfVertices];
    . . .
} VerticesList;
VerticesList Vertices;
```

There are two options for computing data in AoS format: perform operation on the data as it stands in AoS format, or re-arrange it (swizzle it) into SoA format dynamically. See Example 5-21 for code samples of each option based on a dot-product computation.

Example 5-21. AoS and SoA Code Samples

```
; The dot product of an array of vectors (Array) and a fixed vector (Fixed) is a
; common operation in 3D lighting operations, where Array = (x0,y0,z0),(x1,y1,z1),...
; and Fixed = (xF,yF,zF)
; A dot product is defined as the scalar quantity d0 = x0*xF + y0*yF + z0*zF.

;
; AoS code
; All values marked DC are “don’t-care.”

; In the AOS model, the vertices are stored in the xyz format
movaps  xmm0, Array     ; xmm0 = DC, x0, y0, z0
movaps  xmm1, Fixed     ; xmm1 = DC, xF, yF, zF
mulps   xmm0, xmm1      ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhlps xmm, xmm0       ; xmm = DC, DC, DC, x0*xF

addps   xmm1, xmm0      ; xmm0 = DC, DC, DC,
                        ; x0*xF+z0*zFmovaps  xmm2, xmm1
shufps  xmm2, xmm2,55h ; xmm2 = DC, DC,    DC,    y0*yF
addps   xmm2, xmm1      ; xmm1 = DC, DC,    DC,
                        ; x0*xF+y0*yF+z0*zF

; SoA code
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF

movaps xmm0, X          ; xmm0 = x0,x1,x2,x3
movaps xmm1, Y          ; xmm0 = y0,y1,y2,y3
movaps xmm2, Z          ; xmm0 = z0,z1,z2,z3
mulps  xmm0, A          ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps  xmm1, B          ; xmm1 = y0*yF, y1*yF, y2*yF, y3*xF
mulps  xmm2, C          ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps  xmm0, xmm1
addps  xmm0, xmm2       ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
```

Performing SIMD operations on the original AoS format can require more calculations and some operations do not take advantage of all SIMD elements available. Therefore, this option is generally less efficient.

The recommended way for computing data in AoS format is to swizzle each set of elements to SoA format before processing it using SIMD technologies. Swizzling can either be done dynamically during program execution or statically when the data structures are generated. See Chapter 6, “Optimizing for SIMD Integer Applications” and Chapter 7, “Optimizing for SIMD Floating-Point Applications” for examples. Performing the swizzle dynamically is usually better than using AoS, but can be somewhat inefficient because there are extra instructions during

computation. Performing the swizzle statically, when data structures are being laid out, is best as there is no runtime overhead.

As mentioned earlier, the SoA arrangement allows more efficient use of the parallelism of SIMD technologies because the data is ready for computation in a more optimal vertical manner: multiplying components X0,X1,X2,X3 by XF,XF,XF,XF using four SIMD execution slots to produce four unique results. In contrast, computing directly on AoS data can lead to horizontal operations that consume SIMD execution slots but produce only a single scalar result (as shown by the many "don't-care" (DC) slots in Example 5-21).

Use of the SoA format for data structures can lead to more efficient use of caches and bandwidth. When the elements of the structure are not accessed with equal frequency, such as when element x, y, z are accessed ten times more often than the other entries, then SoA saves memory and prevents fetching unnecessary data items a, b, and c.

Example 5-22. Hybrid SoA Data Structure

```
NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];

} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    . . .

} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
```

Note that SoA can have the disadvantage of requiring more independent memory stream references. A computation that uses arrays X, Y, and Z (see Example 5-20) would require three separate data streams. This can require the use of more prefetches, additional address generation calculations, as well as having a greater impact on DRAM page access efficiency.

There is an alternative: a hybrid SoA approach blends the two alternatives (see Example 5-22). In this case, only two separate address streams are generated and referenced:

- One contains XXXX, YYYY,ZZZZ, ZZZZ,… .
- One contains AAAA, BBBB, CCCC, AAAA, DDDD,… .

The approach prevents fetching unnecessary data, assuming the variables X, Y, Z are always used together; whereas the variables A, B, C would also be used together, but not at the same time as X, Y, Z.

The hybrid SoA approach ensures:

- Data is organized to enable more efficient vertical SIMD computation.
- Simpler/less address generation than AoS.
- Fewer streams, which reduces DRAM page misses.
- Use of fewer prefetches, due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

With the advent of the SIMD technologies, the choice of data organization becomes more important and should be carefully based on the operations to be performed on the data.

5.5.2 STRIP-MINING

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure by:

- Increasing the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- Reducing the number of iterations of the loop by a factor of the length of each "vector," or number of operations being performed per SIMD operation. In the case of Intel SSE, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

Consider Example 5-23:

Example 5-23. Pseudo-Code Before Strip Mining

```
typedef struct _VERTEX {
      float x, y, z, nx, ny, nz, u, v;
 } Vertex_rec;

main()
{
      Vertex_rec v[Num];
      ....
      for (i=0; i<Num; i++) {
        Transform(v[i]);
      }
```

```
      for (i=0; i<Num; i++) {
        Lighting(v[i]);
      }
      ....
}
```

The main loop consists of two functions: transformation and lighting. For each object, the main loop calls a transformation routine to update some data, then calls the lighting routine to further work on the data. If the size of array V[NUM] is larger than the cache, then the coordinates for V[I] that were cached during TRANSFORM(V[I]) will be evicted from the cache by the time we do LIGHTING(V[I]). This means that V[I] will have to be fetched from main memory a second time, reducing performance.

In Example 5-24, the computation has been strip-mined to a size STRIP_SIZE. The value STRIP_SIZE is chosen so STRIP_SIZE elements of array V[NUM] fit into the cache hierarchy. By doing this, a given element V[I] brought into

the cache by TRANSFORM(V[I]) will still be in the cache when we perform LIGHTING(V[I]), and thus improve performance over the non-strip-mined code.

Example 5-24. Strip Mined Code

```
MAIN()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i < Num; i+=strip_size) {
      FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
          TRANSFORM(V[J]);
     }
      FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
          LIGHTING(V[J]);
      }
    }
}
```

5.5.3 LOOP BLOCKING

Loop blocking is another useful technique for memory performance optimization. The main purpose of loop blocking is also to eliminate as many cache misses as possible. This technique transforms the memory domain of a given problem into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse. In fact, one can treat loop blocking as strip mining in two or more dimensions.

Consider the code in Example 5-23 and access pattern in Figure 5-5. The two-dimensional array A is referenced in the J (column) direction and then referenced in the I (row) direction (column-major order); whereas array B is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array A and B for the code in Example 5-25 would be 1 and MAX, respectively.

Example 5-25. Loop Blocking

```
A. Original Loop
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
          A[i,j] = A[i,j] + B[j, i];
    }
}
```

Example 5-25. Loop Blocking (Contd.)

```
B. Transformed Loop after Blocking
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```

For the first iteration of the inner loop, each access to array B will generate a cache miss. If the size of one row of array A, that is, A[2, 0:MAX-1], is large enough, by the time the second iteration starts, each access to array B will always generate a cache miss. For instance, on the first iteration, the cache line containing B[0, 0:7] will be brought in when B[0,0] is referenced because the float type variable is four bytes and each cache line is 32 bytes. Due to the limitation of cache capacity, this line will be evicted due to conflict misses before the inner loop reaches the end.

For the next iteration of the outer loop, another cache miss will be generated while referencing B[0, 1]. In this manner, a cache miss occurs when each element of array B is referenced, that is, there is no data reuse in the cache at all for array B.

This situation can be avoided if the loop is blocked with respect to the cache size. In Figure 5-5, a BLOCK_SIZE is selected as the loop blocking factor. Suppose that BLOCK_SIZE is 8, then the blocked chunk of each array will be eight cache lines (32 bytes each). In the first iteration of the inner loop, A[0, 0:7] and B[0, 0:7] will be brought into the cache. B[0, 0:7] will be completely consumed by the first iteration of the outer loop. Consequently, B[0, 0:7] will only experience one cache miss after applying loop blocking optimization in lieu of eight misses for the original algorithm.

As illustrated in Figure 5-5, arrays A and B are blocked into smaller rectangular chunks so that the total size of two blocked A and B chunks is smaller than the cache size. This allows maximum data reuse.

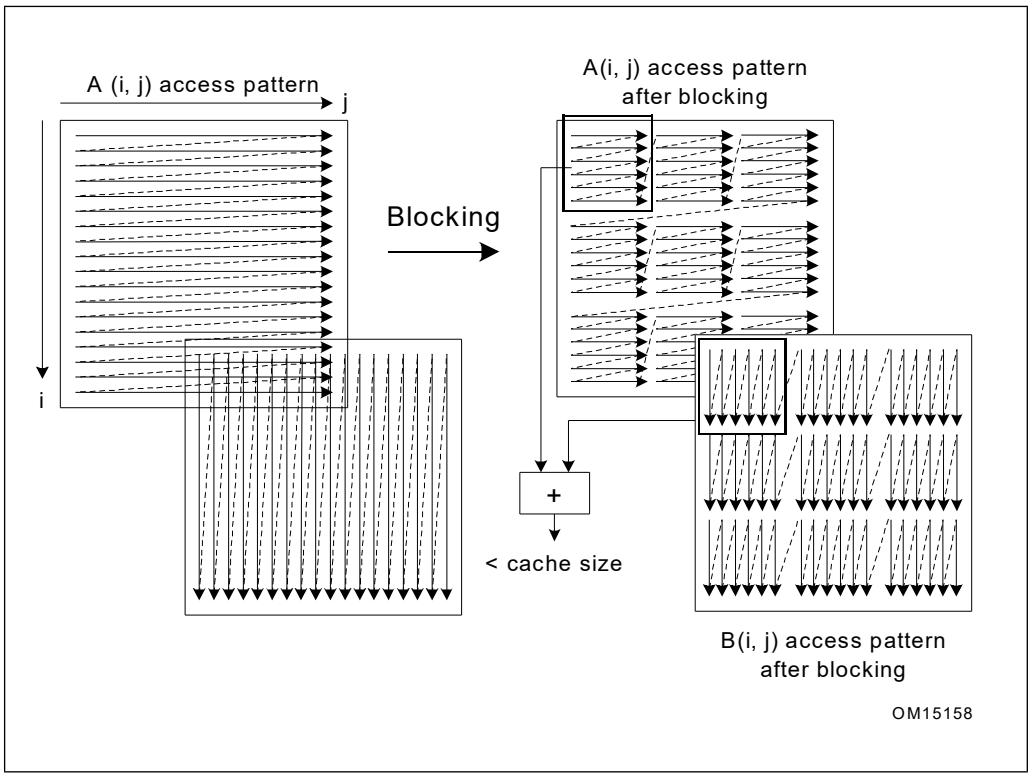


Figure 5-5. Loop Blocking Access Pattern

As one can see, all the redundant cache misses can be eliminated by applying this loop blocking technique. If MAX is huge, loop blocking can also help reduce the penalty from DTLB (data translation look-aside buffer) misses. In addition to improving the cache/memory performance, this optimization technique also saves external bus bandwidth.

5.6 INSTRUCTION SELECTION

The following section gives some guidelines for choosing instructions to complete a task.

One barrier to SIMD computation can be the existence of data-dependent branches. Conditional moves can be used to eliminate data-dependent branches. Conditional moves can be emulated in SIMD computation by using masked

compares and logicals, as shown in Example 5-26. SSE4.1 provides packed blend instruction that can vectorize data-dependent branches in a loop.

Example 5-26. Emulation of Conditional Moves

```
High-level code:
__declspec(align(16)) short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT], D[MAX_ELEMENT],
E[MAX_ELEMENT];

for (i=0; i<MAX_ELEMENT; i++) {
    if (A[i] > B[i]) {
        C[i] = D[i];
    } else {
        C[i] = E[i];
    }

}
MMX assembly code processes 4 short values per iteration:
    xor     eax, eax

top_of_loop:
    movq    mm0, [A + eax]
    pcmpgtwxmm0, [B + eax]; Create compare mask
    movq    mm1, [D + eax]
    pand    mm1, mm0; Drop elements where A<B
    pandn   mm0, [E + eax] ; Drop elements where A>B

    por     mm0, mm1; Crete single word
    movq    [C + eax], mm0
    add     eax, 8
    cmp     eax, MAX_ELEMENT*2
    jle     top_of_loop

SSE4.1 assembly processes 8 short values per iteration:
    xor     eax, eax

top_of_loop:
    movdqq xmm0, [A + eax]
    pcmpgtwxmm0, [B + eax]; Create compare mask
    movdqa xmm1, [E + eax]
    pblendv xmm1, [D + eax], xmm0;
    movdqa [C + eax], xmm1;
    add     eax, 16
    cmp     eax, MAX_ELEMENT*2
    jle     top_of_loop
```

If there are multiple consumers of an instance of a register, group the consumers together as closely as possible. However, the consumers should not be scheduled near the producer.

5.7 TUNING THE FINAL APPLICATION

Once the application is functioning correctly, use a profiler that measures the application while it is running on a system. Intel VTune Amplifier XE can help you determine where to make changes in your application to improve performance. Using Intel VTune Amplifier XE can help you with various phases required for optimized performance. See Appendix A.3.1 for details. After every effort to optimize, you should check the performance gains to see where you are making your major optimization gains.

CHAPTER 6
OPTIMIZING FOR SIMD INTEGER APPLICATIONS

SIMD integer instructions provide performance improvements in applications that are integer-intensive and can take advantage of SIMD architecture.

Guidelines in this chapter for using SIMD integer instructions (in addition to those described in Chapter 3, "General Optimization Guidelines") may be used to develop fast and efficient code that scales across processor generations.

The collection of 64-bit and 128-bit SIMD integer instructions supported by MMX technology, SSE, SSE2, SSE3, SSSE3, SSE4.1, and PCMPEQQ in SSE4.2 are referred to as SIMD integer instructions.

Code sequences in this chapter demonstrates the use of basic 64-bit SIMD integer instructions and more efficient 128-bit SIMD integer instructions.

Processors based on Intel Core microarchitecture support MMX, SSE, SSE2, SSE3, and SSSE3. Processors based on Enhanced Intel Core microarchitecture support SSE4.1 and all previous generations of SIMD integer instructions. Processors based on Nehalem microarchitecture support MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2.

Single-instruction, multiple-data techniques can be applied to text/string processing, lexing and parser applications. SIMD programming in string/text processing and lexing applications often require sophisticated techniques beyond those commonly used in SIMD integer programming. This is covered in Chapter 14, "Intel® SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing."

Execution of 128-bit SIMD integer instructions in Intel Core microarchitecture and Enhanced Intel Core microarchitecture are substantially more efficient than on previous microarchitectures. Thus newer SIMD capabilities introduced in SSE4.1 operate on 128-bit operands and do not introduce equivalent 64-bit SIMD capabilities. Conversion from 64-bit SIMD integer code to 128-bit SIMD integer code is highly recommended.

This chapter contains examples that will help you to get started with coding your application. The goal is to provide simple, low-level operations that are frequently used. The examples use a minimum number of instructions necessary to achieve best performance on the current generation of Intel 64 and IA-32 processors.

Each example includes a short description, sample code, and notes if necessary. These examples do not address scheduling as it is assumed the examples will be incorporated in longer code sequences.

For planning considerations of using the SIMD integer instructions, refer to Section 5.1.3.

6.1 GENERAL RULES ON SIMD INTEGER CODE

General rules and suggestions are:

- Do not intermix 64-bit SIMD integer instructions with x87 floating-point instructions. See Section 6.2 Note that all SIMD integer instructions can be intermixed without penalty.
- Favor 128-bit SIMD integer code over 64-bit SIMD integer code. On microarchitectures prior to Intel Core microarchitecture, most 128-bit SIMD instructions have two-cycle throughput restrictions due to the underlying 64-bit data path in the execution engine. Intel Core microarchitecture executes most SIMD instructions (except shuffle, pack, unpack operations) with one-cycle throughput and provides three ports to execute multiple SIMD instructions in parallel. Enhanced Intel Core microarchitecture speeds up 128-bit shuffle, pack, unpack operations with 1 cycle throughput.
- When writing SIMD code that works for both integer and floating-point data, use the subset of SIMD convert instructions or load/store instructions to ensure that the input operands in XMM registers contain data types that are properly defined to match the instruction.

Code sequences containing cross-typed usage produce the same result across different implementations but incur a significant performance penalty. Using SSE/SSE2/SSE3/SSSE3/SSE44.1 instructions to operate on type-mismatched SIMD data in the XMM register is strongly discouraged.

- Use the optimization rules and guidelines described in Chapter 3 and Chapter 5, "Coding for SIMD Architectures".
- Take advantage of hardware prefetcher where possible. Use the PREFETCH instruction only when data access patterns are irregular and prefetch distance can be pre-determined. See Chapter 9, "Optimizing Cache Usage."
- Emulate conditional moves by using blend, masked compares and logicals instead of using conditional branches.

6.2 USING SIMD INTEGER WITH X87 FLOATING-POINT

All 64-bit SIMD integer instructions use MMX registers, which share register state with the x87 floating-point stack. Because of this sharing, certain rules and considerations apply. Instructions using MMX registers cannot be freely intermixed with x87 floating-point registers. Take care when switching between 64-bit SIMD integer instructions and x87 floating-point instructions to ensure functional correctness. See Section 6.2.1.

Both Section 6.2.1 and Section 6.2.2 apply only to software that employs MMX instructions. As noted before, 128-bit SIMD integer instructions should be favored to replace MMX code and achieve higher performance. That also obviates the need to use EMMS, and the performance penalty of using EMMS when intermixing MMX and X87 instructions.

For performance considerations, there is no penalty of intermixing SIMD floating-point operations and 128-bit SIMD integer operations and x87 floating-point operations.

6.2.1 USING THE EMMS INSTRUCTION

When generating 64-bit SIMD integer code, keep in mind that the eight MMX registers are aliased to x87 floating-point registers. Switching from MMX instructions to x87 floating-point instructions incurs a finite delay, so it is the best to minimize switching between these instruction types. But when switching, the **EMMS** instruction provides an efficient means to clear the x87 stack so that subsequent x87 code can operate properly.

As soon as an instruction makes reference to an MMX register, all valid bits in the x87 floating-point tag word are set, which implies that all x87 registers contain valid values. In order for software to operate correctly, the x87 floating-point stack should be emptied when starting a series of x87 floating-point calculations after operating on the MMX registers.

Using EMMS clears all valid bits, effectively emptying the x87 floating-point stack and making it ready for new x87 floating-point operations. The EMMS instruction ensures a clean transition between using operations on the MMX registers and using operations on the x87 floating-point stack.

Failure to use the EMMS instruction (or the _MM_EMPTY( ) intrinsic) between operations on the MMX registers and x87 floating-point registers may lead to unexpected results.

NOTE

Failure to reset the tag word for FP instructions after using an MMX instruction can result in faulty execution or poor performance.

6.2.2 GUIDELINES FOR USING EMMS INSTRUCTION

When developing code with both x87 floating-point and 64-bit SIMD integer instructions, follow these steps:

1. Always call the EMMS instruction at the end of 64-bit SIMD integer code when the code transitions to x87 floating-point code.

2. Insert the EMMS instruction at the end of all 64-bit SIMD integer code segments to avoid an x87 floating-point stack overflow exception when an x87 floating-point instruction is executed.

When writing an application that uses both floating-point and 64-bit SIMD integer instructions, use the following guidelines to help you determine when to use EMMS:

- **If next instruction is x87 FP:** Use _MM_EMPTY() after a 64-bit SIMD integer instruction if the next instruction is an X87 FP instruction; for example, before doing calculations on floats, doubles or long doubles.
- **Don't empty when already empty:** If the next instruction uses an MMX register, _MM_EMPTY() incurs a cost with no benefit.
- **Group Instructions:** Try to partition regions that use X87 FP instructions from those that use 64-bit SIMD integer instructions. This eliminates the need for an EMMS instruction within the body of a critical loop.
- **Runtime initialization:** Use _MM_EMPTY() during runtime initialization of __M64 and X87 FP data types. This ensures resetting the register between data type transitions. See Example 6-1 for coding usage.

Example 6-1. Resetting Register Between __m64 and FP Data Types Code

Incorrect Usage	Correct Usage
__m64 x = _m_paddd(y, z); float f = init();	__m64 x = _m_paddd(y, z); float f = (_mm_empty(), init());

You must be aware that your code generates an MMX instruction, which uses MMX registers with the Intel C++ Compiler, in the following situations:

- When using a 64-bit SIMD integer intrinsic from MMX technology, SSE/SSE2/SSSE3.
- When using a 64-bit SIMD integer instruction from MMX technology, SSE/SSE2/SSSE3 through inline assembly.
- When referencing the __M64 data type variable.

Additional information on the x87 floating-point programming model can be found in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. For more on EMMS, visit the Intel® C++ Compiler Classic Developer Guide and Reference.

6.3 DATA ALIGNMENT

Make sure that 64-bit SIMD integer data is 8-byte aligned and that 128-bit SIMD integer data is 16-byte aligned. Referencing unaligned 64-bit SIMD integer data can incur a performance penalty due to accesses that span 2 cache lines. Referencing unaligned 128-bit SIMD integer data results in an exception unless the MOVDQU (move double-quadword unaligned) instruction is used. Using the MOVDQU instruction on unaligned data can result in lower performance than using 16-byte aligned references. Refer to Section 5.4 for more information.

Loading 16 bytes of SIMD data efficiently requires data alignment on 16-byte boundaries. SSSE3 provides the PALIGNR instruction. It reduces overhead in situations that requires software to processing data elements from non-aligned address. The PALIGNR instruction is most valuable when loading or storing unaligned data with the address shifts by a few bytes. You can replace a set of unaligned loads with aligned loads followed by using PALIGNR instructions and simple register to register copies.

Using PALIGNRs to replace unaligned loads improves performance by eliminating cache line splits and other penalties. In routines like MEMCPY( ), PALIGNR can boost the performance of misaligned cases. Example 6-2 shows a situation that benefits by using PALIGNR.

Example 6-2. FIR Processing Example in C language Code

```
void FIR(float *in, float *out, float *coeff, int count)
{int i,j;
    for ( i=0; i<count - TAP; i++ )
    {     float sum = 0;
          for ( j=0; j<TAP; j++ )
          {     sum += in[j]*coeff[j]; }
          *out++ = sum;
          in++;
    }
}
```

Example 6-3 compares an optimal SSE2 sequence of the FIR loop and an equivalent SSSE3 implementation. Both implementations unroll 4 iteration of the FIR inner loop to enable SIMD coding techniques. The SSE2 code can not avoid experiencing cache line split once every four iterations. PALGNR allows the SSSE3 code to avoid the delays associated with cache line splits.

Example 6-3. SSE2 and SSSE3 Implementation of FIR Processing Code

Optimized for SSE2	Optimized for SSSE3
pxor xmm0, xmm0 xor ecx, ecx mov eax, dword ptr[input] mov ebx, dword ptr[coeff4] inner_loop: movapssxmm1, xmmword ptr[eax+ecx] mulpsxmm1, xmmword ptr[ebx+4*ecx] addpsxmm0, xmm1 movups xmm1, xmmword ptr[eax+ecx+4] mulps xmm1, xmmword ptr[ebx+4*ecx+16] addps xmm0, xmm1 movups xmm1, xmmword ptr[eax+ecx+8] mulps xmm1, xmmword ptr[ebx+4*ecx+32] addps xmm0, xmm1 movups xmm1, xmmword ptr[eax+ecx+12] mulps xmm1, xmmword ptr[ebx+4*ecx+48] addps xmm0, xmm1	pxor xmm0, xmm0 xor ecx, ecx mov eax, dword ptr[input] mov ebx, dword ptr[coeff4] inner_loop: movapsxmm1, xmmword ptr[eax+ecx] movapsxmm3, xmm1 mulpsxmm1, xmmword ptr[ebx+4*ecx] addpsxmm0, xmm1 movaps xmm2, xmmword ptr[eax+ecx+16] movaps xmm1, xmm2 palignr xmm2, xmm3, 4 mulps xmm2, xmmword ptr[ebx+4*ecx+16] addps xmm0, xmm2 movaps xmm2, xmm1 palignr xmm2, xmm3, 8 mulps xmm2, xmmword ptr[ebx+4*ecx+32] addps xmm0, xmm2

Example 6-3. SSE2 and SSSE3 Implementation of FIR Processing Code (Contd.)

Optimized for SSE2	Optimized for SSSE3

add ecx, 16
cmp ecx, 4*TAP
jl inner_loop
 	
mov eax, dword ptr[output]
movaps xmmword ptr[eax], xmm0
	movaps xmm2, xmm1
	palignr xmm2, xmm3, 12
	mulps xmm2, xmmword ptr[ebx+4*ecx+48]
	addps xmm0, xmm2
	
	add ecx, 16
	cmp ecx, 4*TAP
	jl inner_loop
	
	mov eax, dword ptr[output]
	movaps xmmword ptr[eax], xmm0

6.4 DATA MOVEMENT CODING TECHNIQUES

In general, better performance can be achieved if data is pre-arranged for SIMD computation (see Section 5.5). This may not always be possible.

This section covers techniques for gathering and arranging data for more efficient SIMD computation.

6.4.1 UNSIGNED UNPACK

MMX technology provides several instructions that are used to pack and unpack data in the MMX registers. SSE2 extends these instructions so that they operate on 128-bit source and destinations.

The unpack instructions can be used to zero-extend an unsigned number. Example 6-4 assumes the source is a packed-word (16-bit) data type.

Example 6-4. Zero Extend 16-bit Values into 32 Bits Using Unsigned Unpack Instructions Code

```
; Input:
;             XMM0        8 16-bit values in source
;             XMM7 0      a local variable can be used
;                         instead of the register XMM7 if
;                         desired.
; Output:
;             XMM0        four zero-extended 32-bit
;                         doublewords from four low-end
;                         words

;             XMM1        four zero-extended 32-bit
;                         doublewords from four high-end
;                         words

movdqa      xmm1, xmm0; copy source
punpcklwd   xmm0, xmm7; unpack the 4 low-end words
                        ; into 4 32-bit doubleword
punpckhwd xmm1, xmm7; unpack the 4 high-end words
                        ; into 4 32-bit doublewords
```

6.4.2 SIGNED UNPACK

Signed numbers should be sign-extended when unpacking values. This is similar to the zero-extend shown above, except that the PSRAD instruction (packed shift right arithmetic) is used to sign extend the values.

Example 6-5 assumes the source is a packed-word (16-bit) data type.

Example 6-5. Signed Unpack Code

```
 Input:
;               XMM0        source value
; Output:
;               XMM0        four sign-extended 32-bit doublewords
;                           from four low-end words
;               XMM1        four sign-extended 32-bit doublewords
;                           from four high-end words
;

movdqa      xmm1, xmm0; copy source
punpcklwd   xmm0, xmm0; unpack four low end words of the source
                        ; into the upper 16 bits of each doubleword
                        ; in the destination
punpckhwd xmm1, xmm1; unpack 4 high-end words of the source
                        ; into the upper 16 bits of each doubleword
                        ; in the destination

psrad       xmm0, 16    ; sign-extend the 4 low-end words of the source
                        ; into four 32-bit signed doublewords
psrad       xmm1, 16    ; sign-extend the 4 high-end words of the
                        ; source into four 32-bit signed doublewords
```

6.4.3 INTERLEAVED PACK WITH SATURATION

Pack instructions pack two values into a destination register in a predetermined order. PACKSSDW saturates two signed doublewords from a source operand and two signed doublewords from a destination operand into four signed words; and it packs the four signed words into a destination register. See Figure 6-1.

SSE2 extends PACKSSDW so that it saturates four signed doublewords from a source operand and four signed doublewords from a destination operand into eight signed words; the eight signed words are packed into the destination.

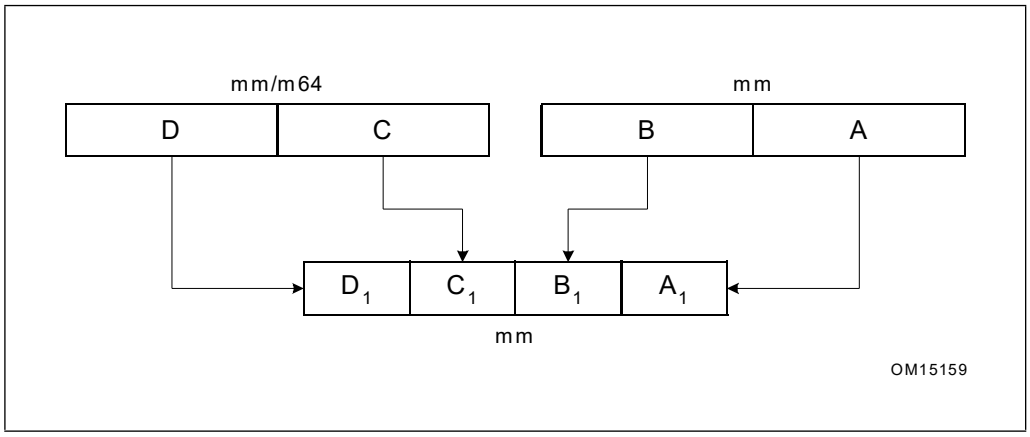


Figure 6-1. PACKSSDW mm, mm/mm64 Instruction

Figure 6-2 illustrates where two pairs of values are interleaved in a destination register; Example 6-6 shows MMX code that accomplishes the operation.

Two signed doublewords are used as source operands and the result is interleaved signed words. The sequence in Example 6-6 can be extended in SSE2 to interleave eight signed words using XMM registers.

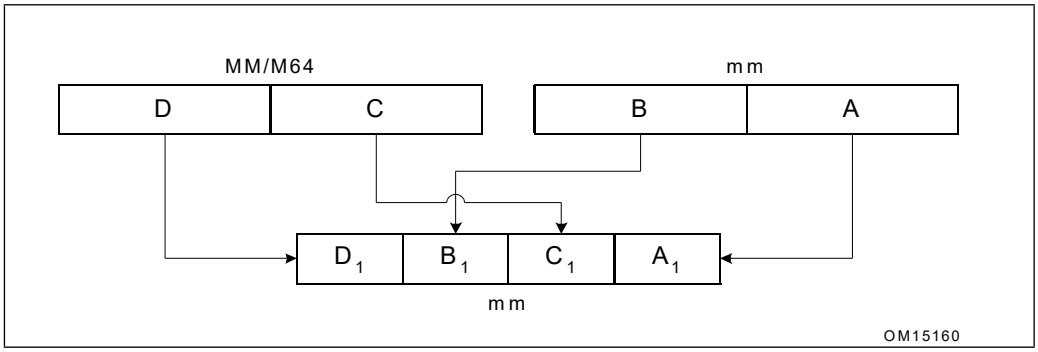


Figure 6-2. Interleaved Pack with Saturation

Example 6-6. Interleaved Pack with Saturation Code

```
; Input:
                MM0     signed source1 value
;               MM1     signed source2 value
; Output:
                MM0     the first and third words contain the
;                       signed-saturated doublewords from MM0,
;                       the second and fourth words contain
;                       signed-saturated doublewords from MM1
;
packssdw        mm0, mm0   ; pack and sign saturate
packssdw        mm1, mm1   ; pack and sign saturate
```

Example 6-6. Interleaved Pack with Saturation Code (Contd.)

```
punpcklwd  mm0, mm1   ; interleave the low-end 16-bit
                      ; values of the operands
```

Pack instructions always assume that source operands are signed numbers. The result in the destination register is always defined by the pack instruction that performs the operation. For example, PACKSSDW packs each of two signed 32-bit values of two sources into four saturated 16-bit signed values in a destination register. PACKUSWB, on the other hand, packs the four signed 16-bit values of two sources into eight saturated eight-bit unsigned values in the destination.

6.4.4 INTERLEAVED PACK WITHOUT SATURATION

Example 6-7 is similar to Example 6-6 except that the resulting words are not saturated. In addition, to protect against overflow, only the low order 16 bits of each doubleword are used. Again, Example 6-7 can be extended in SSE2 to accomplish interleaving eight words without saturation.

Example 6-7. Interleaved Pack without Saturation Code

```
; Input:
;       MM0         signed source value
;       MM1         signed source value

; Output:
;       MM0         the first and third words contain the
;                   low 16-bits of the doublewords in MM0,
;                   the second and fourth words contain the
;                   low 16-bits of the doublewords in MM1

pslld   mm1, 16     ; shift the 16 LSB from each of the
                    ; doubleword values to the 16 MSB
                    ; position
pand    mm0, {0,ffff,0,ffff}
                    ; mask to zero the 16 MSB
                    ; of each doubleword value
por     mm0, mm1    ; merge the two operands
```

6.4.5 NON-INTERLEAVED UNPACK

Unpack instructions perform an interleave merge of the data elements of the destination and source operands into the destination register.

The following example merges the two operands into destination registers without interleaving. For example, take two adjacent elements of a packed-word data type in SOURCE1 and place this value in the low 32 bits of the results. Then take two adjacent elements of a packed-word data type in SOURCE2 and place this value in the high 32 bits of the results. One of the destination registers will have the combination illustrated in Figure 6-3.

The other destination register will contain the opposite combination illustrated in Figure 6-4.

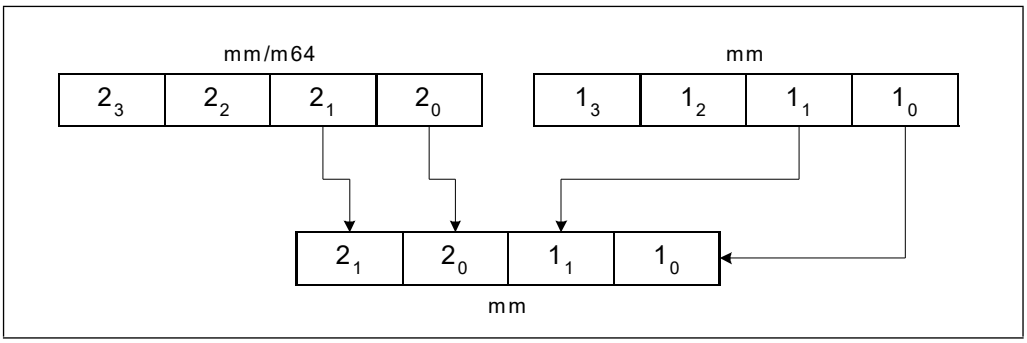


Figure 6-3. Result of Non-Interleaved Unpack Low in MM0

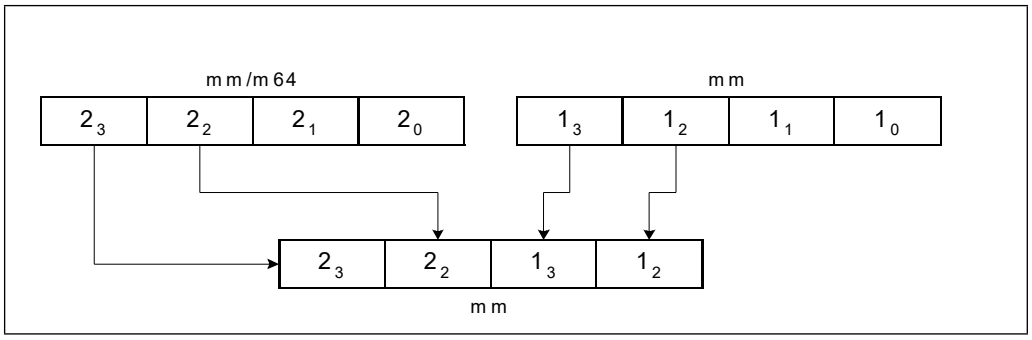


Figure 6-4. Result of Non-Interleaved Unpack High in MM1

Code in the Example 6-8 unpacks two packed-word sources in a non-interleaved way. The goal is to use the instruction which unpacks doublewords to a quadword, instead of using the instruction which unpacks words to doublewords.

Example 6-8. Unpacking Two Packed-word Sources in Non-Interleaved Way Code

```
; Input:
;             MM0          packed-word source value
;             MM1          packed-word source value
; Output:
;             MM0          contains the two low-end words of the
;                          original sources, non-interleaved
;             MM2          contains the two high end words of the
;                          original sources, non-interleaved.
movq        mm2, mm0     ; copy source1
punpckldq   mm0, mm1     ; replace the two high-end words of MMO with
                         ; two low-end words of MM1;
                         ; leave the two low-end words of MM0 in place
punpckhdq   mm2, mm1     ; move two high-end words of MM2 to the two low-end
                         ; words of MM2; place the two high-end words of
                         ; MM1 in two high-end words of MM2
```

6.4.6 EXTRACT DATA ELEMENT

The PEXTRW instruction in SSE takes the word in the designated MMX register selected by the two least significant bits of the immediate value and moves it to the lower half of a 32-bit integer register. See Figure 6-5 and Example 6-9.

With SSE2, PEXTRW can extract a word from an XMM register to the lower 16 bits of an integer register. SSE4.1 provides extraction of a byte, word, dword and qword from an XMM register into either a memory location or integer register.

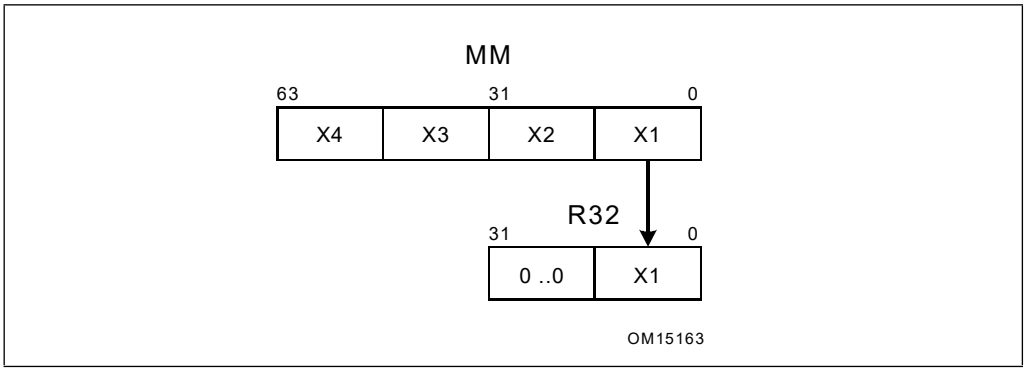


Figure 6-5. PEXTRW Instruction

Example 6-9. PEXTRW Instruction Code

```
; Input:
;         eax       source value
;                   immediate value: “0”
; Output:
;         edx       32-bit integer register containing the extracted word in the
;                    low-order bits & the high-order bits zero-extended
movq    mm0, [eax]
pextrw edx, mm0, 0
```

6.4.7 INSERT DATA ELEMENT

The PINSRW instruction in SSE loads a word from the lower half of a 32-bit integer register or from memory and inserts it in an MMX technology destination register at a position defined by the two least significant bits of the immediate constant. Insertion is done in such a way that three other words from the destination register are left untouched. See Figure 6-6 and Example 6-10.

With SSE2, PINSRW can insert a word from the lower 16 bits of an integer register or memory into an XMM register. SSE4.1 provides insertion of a byte, dword and qword from either a memory location or integer register into an XMM register.

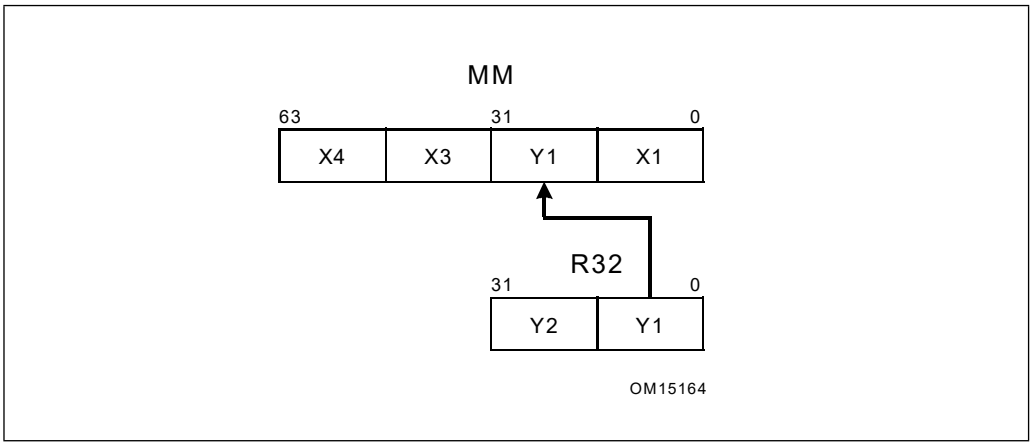


Figure 6-6. PINSRW Instruction

Example 6-10. PINSRW Instruction Code

```
; Input:
;         edx       pointer to source value
; Output:
;         mm0 register with new 16-bit value inserted
;
mov     eax, [edx]
pinsrw  mm0, eax, 1
```

If all of the operands in a register are being replaced by a series of PINSRW instructions, it can be useful to clear the content and break the dependence chain by either using the PXOR instruction or loading the register. See Example 6-11 and Section 3.5.1.7

Example 6-11. Repeated PINSRW Instruction Code

```
; Input:
;               edx             pointer to structure containing source
;                               values at offsets: of +0, +10, +13, and +24
;                               immediate value: “1”
; Output:
;               MMX             register with new 16-bit value inserted
;
pxor            mm0, mm0        ; Breaks dependency on previous value of mm0
mov             eax, [edx]
pinsrw          mm0, eax, 0
mov             eax, [edx+10]
pinsrw          mm0, eax, 1
mov             eax, [edx+13]
pinsrw          mm0, eax, 2
mov             eax, [edx+24]
pinsrw          mm0, eax, 3
```

6.4.8 NON-UNIT STRIDE DATA MOVEMENT

SSE4.1 provides instructions to insert a data element from memory into an XMM register, and to extract a data element from an XMM register into memory directly. Separate instructions are provided to handle floating-point data and integer byte, word, or dword. These instructions are suited for vectorizing code that loads/stores non-unit stride data from memory, see Example 6-12.

Example 6-12. Non-Unit Stride Load/Store Using SSE4.1 Instructions

Goal: Non-Unit Stride Load Dwords	Goal: Non-Unit Stride Store Dwords
movd xmm0, [addr] pinsrd xmm0, [addr + stride], 1 pinsrd xmm0, [addr + 2*stride], 2 pinsrd xmm0, [addr + 3*stride], 3	movd [addr], xmm0 pextrd [addr + stride], xmm0, 1 pextrd [addr + 2*stride], xmm0, 2 pextrd [addr + 3*stride], xmm0, 3

Example 6-13 provides two examples: using INSERTPS and PEXTRD to perform gather operations on floating-point data; using EXTRACTPS and PEXTRD to perform scatter operations on floating-point data.

Example 6-13. Scatter and Gather Operations Using SSE4.1 Instructions

Goal: Gather Operation	Goal: Scatter Operation
movd eax, xmm0 movss xmm1, [addr + 4*eax] pextrd eax, xmm0, 1 insertps xmm1, [addr + 4*eax], 1 pextrd eax, xmm0, 2 insertps xmm1, [addr + 4*eax], 2 pextrd eax, xmm0, 3 insertps xmm1, [addr + 4*eax], 3	movd eax, xmm0 movss [addr + 4*eax], xmm1 pextrd eax, xmm0, 1 extractps [addr + 4*eax], xmm1, 1 pextrd eax, xmm0, 2 extractps [addr + 4*eax], xmm1, 2 pextrd eax, xmm0, 3 extractps [addr + 4*eax], xmm1, 3

6.4.9 MOVE BYTE MASK TO INTEGER

The PMOVMSKB instruction returns a bit mask formed from the most significant bits of each byte of its source operand. When used with 64-bit MMX registers, this produces an 8-bit mask, zeroing out the upper 24 bits in the destination register. When used with 128-bit XMM registers, it produces a 16-bit mask, zeroing out the upper 16 bits in the destination register.

The 64-bit version of this instruction is shown in Figure 6-7 and Example 6-14.

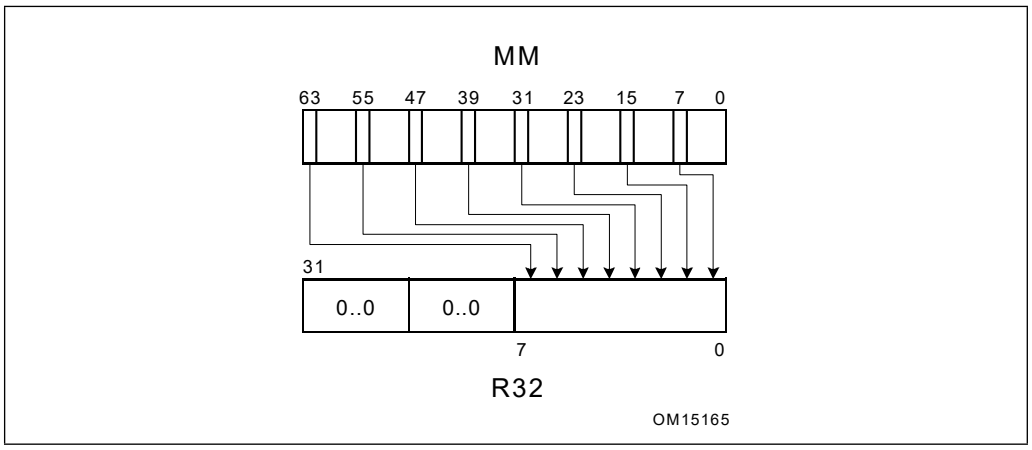


Figure 6-7. PMOVSMKB Instruction

Example 6-14. PMOVMSKB Instruction Code

```
; Input:
;          source value
; Output:
;          32-bit register containing the byte mask in the lower eight bits
;
movq     mm0, [edi]
pmovmskb eax, mm0
```

6.4.10 PACKED SHUFFLE WORD FOR 64-BIT REGISTERS

The PSHUFW instruction uses the immediate (IMM8) operand to select between the four words in either two MMX registers or one MMX register and a 64-bit memory location. SSE2 provides PSHUFLW to shuffle the lower four words into an XMM register. In addition to the equivalent to the PSHUFW, SSE2 also provides PSHUFHW to shuffle the higher four words. Furthermore, SSE2 offers PSHUFD to shuffle four dwords into an XMM register. All of these four PSHUF instructions use an immediate byte to encode the data path of individual words within the corresponding 8 bytes from source to destination, shown in Table 6-1.

Table 6-1. PSHUF Instruction Byte Encoding of Individual Words

Bits	Words
1 - 0	0
3 - 2	1
5 - 4	2
7 - 6	3

6.4.11 PACKED SHUFFLE WORD FOR 128-BIT REGISTERS

The PSHUFLW/PSHUFHW instruction performs a full shuffle of any source word field within the low/high 64 bits to any result word field in the low/high 64 bits, using an 8-bit immediate operand; other high/low 64 bits are passed through from the source operand.

PSHUFD performs a full shuffle of any double-word field within the 128-bit source to any double-word field in the 128-bit result, using an 8-bit immediate operand.

No more than 3 instructions, using PSHUFLW/PSHUFHW/PSHUFD, are required to implement many common data shuffling operations. Broadcast, Swap, and Reverse are illustrated in Example 6-15 and Example 6-16.

Example 6-15. Broadcast a Word Across XMM, Using 2 SSE2 Instructions

```
/* Goal:  Broadcast the value from word 5 to all words */
/* Instruction     Result */
                    | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFHW (3,2,1,1)| 7| 6| 5| 5| 3| 2| 1| 0|

PSHUFD (2,2,2,2)| 5| 5| 5| 5| 5| 5| 5| 5|
```

Example 6-16. Swap/Reverse Words in an XMM, Using 3 SSE2 Instructions

Goal: Swap the values in word 6 and word 1	Goal: Reverse the order of the words
/* Instruction Result */ | 7| 6| 5| 4| 3| 2| 1| 0|	/* Instruction Result */ | 7| 6| 5| 4| 3| 2| 1| 0|
PSHUFD (3,0,1,2)| 7| 6| 1| 0| 3| 2| 5| 4|	PSHUFLW (0,1,2,3)| 7| 6| 5| 4| 0| 1| 2| 3|
PSHUFHW (3,1,2,0)| 7| 1| 6| 0| 3| 2| 5| 4|	PSHUFHW (0,1,2,3)| 4| 5| 6| 7| 0| 1| 2| 3|
PSHUFD (3,0,1,2)| 7| 1| 5| 4| 3| 2| 6| 0|	PSHUFD (1,0,3,2)| 0| 1| 2| 3| 4| 5| 6| 7|

6.4.12 SHUFFLE BYTES

SSSE3 provides PSHUFB; this instruction carries out byte manipulation within a 16 byte range. PSHUFB can replace up to 12 other instructions: including SHIFT, OR, AND and MOV.

Use PSHUFB if the alternative uses 5 or more instructions.

6.4.13 CONDITIONAL DATA MOVEMENT

SSE4.1 provides two packed blend instructions on byte and word data elements in 128-bit operands. Packed blend instructions conditionally copies data elements from selected positions in the source to the corresponding data element using a mask specified by an immediate control byte or an implied XMM register (XMM0). The mask can be generated by a packed compare instruction for example. Thus packed blend instructions are most useful for

vectorizing conditional flows within a loop and can be more efficient than inserting single element one at a time for some situations.

6.4.14 UNPACKING/INTERLEAVING 64-BIT DATA IN 128-BIT REGISTERS

The PUNPCKLQDQ/PUNPCHQDQ instructions interleave the low/high-order 64-bits of the source operand and the low/high-order 64-bits of the destination operand. It then writes the results to the destination register.

The high/low-order 64-bits of the source operands are ignored.

6.4.15 DATA MOVEMENT

There are two additional instructions to enable data movement from 64-bit SIMD integer registers to 128-bit SIMD registers.

The MOVQ2DQ instruction moves the 64-bit integer data from an MMX register (source) to a 128-bit destination register. The high-order 64 bits of the destination register are zeroed-out.

The MOVDQ2Q instruction moves the low-order 64-bits of integer data from a 128-bit source register to an MMX register (destination).

6.4.16 CONVERSION INSTRUCTIONS

SSE provides Instructions to support 4-wide conversion of single-precision data to/from double-word integer data. Conversions between double-precision data to double-word integer data have been added in SSE2.

SSE4.1 provides 4 rounding instructions to convert floating-point values to integer values with rounding control specified in a more flexible manner and independent of the rounding control in MXCSR. The integer values produced by ROUNDxx instructions are maintained as floating-point data.

SSE4.1 also provides instructions to convert integer data from:

- Packed bytes to packed word/dword/qword format using either sign extension or zero extension.
- Packed words to packed dword/qword format using either sign extension or zero extension.
- Packed dword to packed qword format using either sign extension or zero extension.

6.5 GENERATING CONSTANTS

SIMD integer instruction sets do not have instructions that will load immediate constants to the SIMD registers.

The following code segments generate frequently used constants in the SIMD register. These examples can also be extended in SSE2 by substituting MMX with XMM registers. See Example 6-17.

Example 6-17. Generating Constants

```
pxor     mm0, mm0   ; generate a zero register in MM0
pcmpeqmm1, mm1   ; Generate all 1's in register MM1,
                   ; which is -1 in each of the packed
                   ; data type fields
```

Example 6-17. Generating Constants (Contd.)

```
pxor     mm0, mm0
pcmpeqmm1, mm1
psubb   mm0, mm1 [psubw  mm0, mm1] (psubd  mm0, mm1)
                        ; three instructions above generate
                        ; the constant 1 in every
                        ; packed-byte [or packed-word]
                        ; (or packed-dword) field
pcmpeqmm1, mm1
psrlw    mm1, 16-n(psrld  mm1, 32-n)
                        ; two instructions above generate
                        ; the signed constant 2^n–1 in every
                        ; packed-word (or packed-dword) field
pcmpeqmm1, mm1
psllw    mm1, n  (pslld mm1, n)
                        ; two instructions above generate
                        ; the signed constant -2n in every
                        ; packed-word (or packed-dword) field
```

NOTE

Because SIMD integer instruction sets do not support shift instructions for bytes, 2n–1 and -2n are relevant only for packed words and packed doublewords.

6.6 BUILDING BLOCKS

This section describes instructions and algorithms which implement common code building blocks.

6.6.1 ABSOLUTE DIFFERENCE OF UNSIGNED NUMBERS

Example 6-18 computes the absolute difference of two unsigned numbers. It assumes an unsigned packed-byte data type.

Here, we make use of the subtract instruction with unsigned saturation. This instruction receives UNSIGNED operands and subtracts them with UNSIGNED saturation. This support exists only for packed bytes and packed words, not for packed doublewords.

Example 6-18. Absolute Difference of Two Unsigned Numbers

```
; Input:
;         MM0 source operand
;         MM1 source operand
; Output:
;         MM0 absolute difference of the unsigned operands

movq    mm2, mm0   ; make a copy of mm0
psubusbmm0, mm1  ; compute difference one way
psubusbmm1, mm2  ; compute difference the other way
por      mm0, mm1   ; OR them together
```

This example will not work if the operands are signed. Note that PSADBW may also be used in some situations. See Section 6.6.9 for details.

6.6.2 ABSOLUTE DIFFERENCE OF SIGNED NUMBERS

Example 6-19 computes the absolute difference of two signed numbers using SSSE3 instruction PABSW. This sequence is more efficient than using previous generation of SIMD instruction extensions.

Example 6-19. Absolute Difference of Signed Numbers

```
;Input:
;       XMM0 signed source operand
;       XMM1 signed source operand

;Output:
;       XMM1absolute difference of the unsigned operands

psubw  xmm0, xmm1; subtract words
pabsw  xmm1, xmm0; results in XMM1
```

6.6.3 ABSOLUTE VALUE

Example 6-20 show an MMX code sequence to compute | X | , where X is signed. This example assumes signed words to be the operands.

With SSSE3, this sequence of three instructions can be replaced by the PABSW instruction. Additionally, SSSE3 provides a 128-bit version using XMM registers and supports byte, word and doubleword granularity.

Example 6-20. Computing Absolute Value

```
; Input:
;       MM0         signed source operand
; Output:
;       MM1         ABS(MMO)
pxor    mm1, mm1    ; set mm1 to all zeros
psubw   mm1, mm0    ; make each mm1 word contain the
                    ; negative of each mm0 word
pmaxswmm1, mm0      ; mm1 will contain only the positive
                    ; (larger) values - the absolute value
```

NOTE

The absolute value of the most negative number (that is, 8000H for 16-bit) cannot be represented using positive numbers. This algorithm will return the original value for the absolute value (8000H).

6.6.4 PIXEL FORMAT CONVERSION

SSSE3 provides the PSHUFB instruction to carry out byte manipulation within a 16-byte range. PSHUFB can replace a set of up to 12 other instruction, including SHIFT, OR, AND and MOV.

Use PSHUFB if the alternative code uses 5 or more instructions. Example 6-21 shows the basic form of conversion of color pixel formats.

Example 6-21. Basic C Implementation of RGBA to BGRA Conversion

```
Standard C Code:
struct RGBA{BYTE r,g,b,a;};
struct BGRA{BYTE b,g,r,a;};

void BGRA_RGBA_Convert(BGRA *source, RGBA *dest, int num_pixels)
{
    for(int i = 0; i < num_pixels; i++){
        dest[i].r = source[i].r;
        dest[i].g = source[i].g;
        dest[i].b = source[i].b;
        dest[i].a = source[i].a;
    }
}
```

Example 6-22 and Example 6-23 show SSE2 code and SSSE3 code for pixel format conversion. In the SSSE3 example, PSHUFB replaces six SSE2 instructions.

Example 6-22. Color Pixel Format Conversion Using SSE2

```
; Optimized for SSE2

mov    esi, src
    mov    edi, dest
    mov ecx, iterations
    movdqa xmm0, ag_mask //{0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff}
    movdqa xmm5, rb_mask //{ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0,ff,0}
    mov eax, remainder

convert16Pixs: // 16 pixels, 64 byte per iteration
    movdqa xmm1, [esi] // xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
    movdqa xmm2, xmm1
    movdqa xmm7, xmm1      //xmm7 abgr
    psrld  xmm2, 16        //xmm2 00ab
    pslld  xmm1, 16        //xmm1 gr00


    por    xmm1, xmm2      //xmm1 grab
    pand   xmm7, xmm0      //xmm7 a0g0
    pand   xmm1, xmm5      //xmm1 0r0b
    por    xmm1, xmm7      //xmm1 argb
    movdqa [edi], xmm1

    //repeats for another 3*16 bytes
    …
```

Example 6-22. Color Pixel Format Conversion Using SSE2 (Contd.)

```
    add  esi, 64
    add  edi, 64
    sub  ecx, 1
    jnz  convert16Pixs
```

Example 6-23. Color Pixel Format Conversion Using SSSE3

```
; Optimized for SSSE3

mov     esi, src
     mov     edi, dest
     mov  ecx, iterations
     movdqa  xmm0, _shufb
// xmm0 = [15,12,13,14,11,8,9,10,7,4,5,6,3,0,1,2]
     mov       eax, remainder

convert16Pixs: // 16 pixels, 64 byte per iteration
     movdqa xmm1, [esi]
// xmm1 = [r3g3b3a3,r2g2b2a2,r1g1b1a1,r0g0b0a0]
     movdqa xmm2, [esi+16]
     pshufb xmm1, xmm0
// xmm1 = [b3g3r3a3,b2g2r2a2,b1g1r1a1,b0g0r0a0]
     movdqa [edi], xmm1

     //repeats for another 3*16 bytes

     …

     add       esi, 64
     add       edi, 64
     sub       ecx, 1
     jnz       convert16Pixs
```

6.6.5 ENDIAN CONVERSION

The PSHUFB instruction can also be used to reverse byte ordering within a doubleword. It is more efficient than traditional techniques, such as BSWAP.

Example 6-24 (a) shows the traditional technique using four BSWAP instructions to reverse the bytes within a DWORD. Each BSWAP requires executing two micro-ops. In addition, the code requires 4 loads and 4 stores for processing 4 DWORDs of data.

Example 6-24 (b) shows an SSSE3 implementation of endian conversion using PSHUFB. The reversing of four DWORDs requires one load, one store, and PSHUFB.

On Intel Core microarchitecture, reversing 4 DWORDs using PSHUFB can be approximately twice as fast as using BSWAP.

Example 6-24. Big-Endian to Little-Endian Conversion

```
;;(a) Using BSWAP
lea eax, src
    lea ecx, dst
    mov edx, elCount
start:
    mov edi, [eax]
    mov esi, [eax+4]
    bswap edi
    mov ebx, [eax+8]

    bswap esi
    mov ebp, [eax+12]
    mov [ecx], edi
    mov [ecx+4], esi
    bswap ebx
    mov [ecx+8], ebx
    bswap ebp
    mov [ecx+12], ebp

    add eax, 16
    add ecx, 16
    sub edx, 4
    jnz start
```

```
;; (b) Using PSHUFB
__declspec(align(16)) BYTE bswapMASK[16] =
{3,2,1,0, 7,6,5,4, 11,10,9,8, 15,14,13,12};
    lea eax, src
    lea ecx, dst
    mov edx, elCount
    movaps   xmm7, bswapMASK
start:
    movdqa   xmm0, [eax]

    pshufb   xmm0, xmm7
    movdqa   [ecx], xmm0
    add eax, 16
    add ecx, 16
    sub edx, 4
    jnz start
```

6.6.6 CLIPPING TO AN ARBITRARY RANGE [HIGH, LOW]

This section explains how to clip a values to a range [HIGH, LOW]. Specifically, if the value is less than LOW or greater than HIGH, then clip to LOW or HIGH, respectively. This technique uses the packed-add and packed-subtract instructions with saturation (signed or unsigned), which means that this technique can only be used on packed-byte and packed-word data types.

The examples in this section use the constants PACKED_MAX and PACKED_MIN and show operations on word values. For simplicity, we use the following constants (corresponding constants are used in case the operation is done on byte values):

- PACKED_MAX equals 0X7FFF7FFF7FFF7FFF
- PACKED_MIN equals 0X8000800080008000
- PACKED_LOW contains the value LOW in all four words of the packed-words data type
- PACKED_HIGH contains the value HIGH in all four words of the packed-words data type
- PACKED_USMAX all values equal 1
- HIGH_US adds the HIGH value to all data elements (4 words) of PACKED_MIN
- LOW_US adds the LOW value to all data elements (4 words) of PACKED_MIN

6.6.6.1 Highly Efficient Clipping

For clipping signed words to an arbitrary range, the PMAXSW and PMINSW instructions may be used. For clipping unsigned bytes to an arbitrary range, the PMAXUB and PMINUB instructions may be used.

Example 6-25 shows how to clip signed words to an arbitrary range; the code for clipping unsigned bytes is similar.

Example 6-25. Clipping to a Signed Range of Words [High, Low]

```
; Input:
;         MM0     signed source operands
; Output:
;         MM0     signed words clipped to the signed
;                 range [high, low]
pminswmm0, packed_high
pmaxswmm0, packed_low
```

With SSE4.1, Example 6-25 can be easily extended to clip signed bytes, unsigned words, signed and unsigned dwords.

Example 6-26. Clipping to an Arbitrary Signed Range [High, Low]

```
; Input:
;         MM0                   signed source operands
; Output:
;         MM1                   signed operands clipped to the unsigned
;                               range [high, low]
paddw  mm0, packed_min      ; add with no saturation
                            ; 0x8000 to convert to unsigned
padduswmm0, (packed_usmax - high_us)
                            ; in effect this clips to high
psubuswmm0, (packed_usmax - high_us + low_us)
                            ; in effect this clips to low
paddw  mm0, packed_low      ; undo the previous two offsets
```

The code above converts values to unsigned numbers first and then clips them to an unsigned range. The last instruction converts the data back to signed data and places the data within the signed range.

Conversion to unsigned data is required for correct results when (High - Low) < 0X8000. If (High - Low) >= 0X8000, simplify the algorithm as in Example 6-27.

Example 6-27. Simplified Clipping to an Arbitrary Signed Range

```
; Input:      MM0          signed source operands
; Output:     MM1          signed operands clipped to the unsigned
;                          range [high, low]
paddssw     mm0, (packed_max - packed_high)
                          ; in effect this clips to high
psubssw     mm0, (packed_usmax - packed_high + packed_low)
                          ; clips to low
paddw       mm0, low      ; undo the previous two offsets
```

This algorithm saves a cycle when it is known that (High - Low) >= 0x8000. The three-instruction algorithm does not work when (High - Low) < 0x8000 because 0xffff minus any number < 0x8000 will yield a number greater in magnitude than 0x8000 (which is a negative number).

When the second instruction, psubssw MM0, (0xffff - High + Low) in the three-step algorithm (Example 6-27) is executed, a negative number is subtracted. The result of this subtraction causes the values in MM0 to be increased instead of decreased, as should be the case, and an incorrect answer is generated.

6.6.6.2 Clipping to an Arbitrary Unsigned Range [High, Low]

Example 6-28 clips an unsigned value to the unsigned range [High, Low]. If the value is less than low or greater than high, then clip to low or high, respectively. This technique uses the packed-add and packed-subtract instructions with unsigned saturation, thus the technique can only be used on packed-bytes and packed-words data types.

Figure 6-28 illustrates operation on word values.

Example 6-28. Clipping to an Arbitrary Unsigned Range [High, Low]

```
; Input:
;             MM0     unsigned source operands
; Output:
;             MM1     unsigned operands clipped to the unsigned
;                     range [HIGH, LOW]
paddusw     mm0, 0xffff - high
                    ; in effect this clips to high
psubusw     mm0, (0xffff - high + low)
                    ; in effect this clips to low
paddw       mm0, low
                    ; undo the previous two offsets
```

6.6.7 PACKED MAX/MIN OF BYTE, WORD AND DWORD

The PMAXSW instruction returns the maximum between four signed words in either of two SIMD registers, or one SIMD register and a memory location.

The PMINSW instruction returns the minimum between the four signed words in either of two SIMD registers, or one SIMD register and a memory location.

The PMAXUB instruction returns the maximum between the eight unsigned bytes in either of two SIMD registers, or one SIMD register and a memory location.

The PMINUB instruction returns the minimum between the eight unsigned bytes in either of two SIMD registers, or one SIMD register and a memory location.

SSE2 extended PMAXSW/PMAXUB/PMINSW/PMINUB to 128-bit operations. SSE4.1 adds 128-bit operations for signed bytes, unsigned word, signed and unsigned dword.

6.6.8 PACKED MULTIPLY INTEGERS

The PMULHUW/PMULHW instruction multiplies the unsigned/signed words in the destination operand with the unsigned/signed words in the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand. The PMULLW instruction multiplies the signed words in the destination operand with the signed words in the source operand. The low-order 16 bits of the 32-bit intermediate results are written to the destination operand.

SSE2 extended PMULHUW/PMULHW/PMULLW to 128-bit operations and adds PMULUDQ.

The PMULUDQ instruction performs an unsigned multiply on the lower pair of double-word operands within 64-bit chunks from the two sources; the full 64-bit result from each multiplication is returned to the destination register.

This instruction is added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

SSE4.1 adds 128-bit operations of PMULDQ and PMULLD. The PMULLD instruction multiplies the signed dwords in the destination operand with the signed dwords in the source operand. The low-order 32 bits of the 64-bit intermediate results are written to the destination operand. The PMULDQ instruction multiplies the two low-order, signed dwords in the destination operand with the two low-order, signed dwords in the source operand and stores two 64-bit results in the destination operand.

6.6.9 PACKED SUM OF ABSOLUTE DIFFERENCES

The PSADBW instruction computes the absolute value of the difference of unsigned bytes for either two SIMD registers, or one SIMD register and a memory location. The differences of 8 pairs of unsigned bytes are then summed to produce a word result in the lower 16-bit field, and the upper three words are set to zero. With SSE2, PSADBW is extended to compute two word results.

The subtraction operation presented above is an absolute difference. That is, T = ABS(X-Y). Byte values are stored in temporary space, all values are summed together, and the result is written to the lower word of the destination register.

Motion estimation involves searching reference frames for best matches. Sum absolute difference (SAD) on two blocks of pixels is a common ingredient in video processing algorithms to locate matching blocks of pixels. PSADBW can be used as building blocks for finding best matches by way of calculating SAD results on 4x4, 8x4, 8x8 blocks of pixels.

6.6.10 MPSADBW AND PHMINPOSUW

The MPSADBW instruction in SSE4.1 performs eight SAD operations. Each SAD operation produces a word result from 4 pairs of unsigned bytes. With 8 SAD result in an XMM register, PHMINPOSUM can help search for the best match between eight 4x4 pixel blocks.

For motion estimation algorithms, MPSADBW is likely to improve over PSADBW in several ways:

- Simplified data movement to construct packed data format for SAD computation on pixel blocks.
- Higher throughput in terms of SAD results per iteration (less iteration required per frame).
- MPSADBW results are amenable to efficient search using PHMINPOSUW.

Examples of MPSADBW vs. PSADBW for 4x4 and 8x8 block search can be found in the white paper listed in the reference section of Chapter 1, “Introduction”.

6.6.11 PACKED AVERAGE (BYTE/WORD)

The PAVGB and PAVGW instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register, along with a carry-in. The results of the addition are then independently shifted to the right by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

The destination operand is an SIMD register. The source operand can either be an SIMD register or a memory operand.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

6.6.12 COMPLEX MULTIPLY BY A CONSTANT

Complex multiplication is an operation which requires four multiplications and two additions. This is exactly how the PMADDWD instruction operates. to use this instruction, you need to format the data into multiple 16-bit values. The real and imaginary components should be 16-bits each. Consider Example 6-29, which assumes that the 64-bit MMX registers are being used:

- Let the input data be DR and DI, where DR is real component of the data and DI is imaginary component of the data.
- Format the constant complex coefficients in memory as four 16-bit values [CR -CI CI CR]. Remember to load the values into the MMX register using MOVQ.
- The real component of the complex product is PR = DR*CR - DI*CI and the imaginary component of the complex product is PI = DR*CI + DI*CR.
- The output is a packed doubleword. If needed, a pack instruction can be used to convert the result to 16-bit (thereby matching the format of the input).

Example 6-29. Complex Multiply by a Constant

```
; Input:
;             MM0          complex value, Dr, Di
;             MM1          constant complex coefficient in the form
;                          [Cr -Ci Ci Cr]
; Output:
;             MM0          two 32-bit dwords containing [Pr Pi]
;
punpckldq   mm0, mm0     ; makes [dr di dr di]
pmaddwd     mm0, mm1     ; done, the result is
                         ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]
```

6.6.13 PACKED 64-BIT ADD/SUBTRACT

The PADDQ/PSUBQ instructions add/subtract quad-word operands within each 64-bit chunk from the two sources; the 64-bit result from each computation is written to the destination register. Like the integer ADD/SUB instruction, PADDQ/PSUBQ can operate on either unsigned or signed (two's complement notation) integer operands.

When an individual result is too large to be represented in 64-bits, the lower 64-bits of the result are written to the destination operand and therefore the result wraps around. These instructions are added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

6.6.14 128-BIT SHIFTS

The PSLLDQ/PSRLDQ instructions shift the first operand to the left/right by the number of bytes specified by the immediate operand. The empty low/high-order bytes are cleared (set to zero).

If the value specified by the immediate operand is greater than 15, then the destination is set to all zeros.

6.6.15 PTEST AND CONDITIONAL BRANCH

SSE4.1 offers PTEST instruction that can be used in vectorizing loops with conditional branches. PTEST is an 128-bit version of the general-purpose instruction TEST. The ZF or CF field of the EFLAGS register are modified as a result of PTEST.

Example 6-30(a) depicts a loop that requires a conditional branch to handle the special case of divide-by-zero. to vectorize such loop, any iteration that may encounter divide-by-zero must be treated outside the vectorizable iterations.

Example 6-30. Using PTEST to Separate Vectorizable and Non-Vectorizable Loop Iterations

(a) Loops Requiring Infrequent Exception Handling	(b) PTEST Enables Early Out to Handle Infrequent, Non-Vectorizable Portion

float a[CNT];
unsigned int i;
 	
for (i=0;i<CNT;i++)
{
if (a[i] != 0.0)
{ a[i] = 1.0f/a[i];
}
else
{ call DivException();
}
}
	xor eax,eax
	movaps xmm7, [all_ones]
	xorps xmm6, xmm6
	lp:
	movaps xmm0, a[eax]
	cmpeqps xmm6, xmm0 ; convert each non-zero to ones
	ptest xmm6, xmm7
	jnc zero_present; carry will be set if all 4 were non-zero
	movaps xmm1,[_1_0f_]
	divps xmm1, xmm0
	movaps a[eax], xmm1
	add eax, 16
	cmp eax, CNT
	jnz lp
	jmp end
	zero_present:
	// execute one by one, call
	// exception when value is zero

Example 6-30(b) shows an assembly sequence that uses PTEST to cause an early-out branch whenever any one of the four floating-point values in xmm0 is zero. The fall-through path enables the rest of the floating-point calculations to be vectorized because none of the four values are zero.

6.6.16 VECTORIZATION OF HETEROGENEOUS COMPUTATIONS ACROSS LOOP ITERATIONS

Vectorization techniques on unrolled loops generally rely on repetitive, homogeneous operations between each loop iteration. Using variable blend instructions, vectorization of heterogeneous operations across loop iterations may be possible.

Example 6-31(a) depicts a simple heterogeneous loop. The heterogeneous operation and conditional branch makes simple loop-unrolling techniques infeasible for vectorization.

Example 6-31. Using Variable BLEND to Vectorize Heterogeneous Loops

(a) Loops with Heterogeneous Operation Across Iterations	(b) Vectorize Condition Flow with BLENDVPS

float a[CNT];
unsigned int i;
 	
for (i=0;i<CNT;i++)
{
  if (a[i] > b[i])
  { a[i] += b[i]; }
  else
  { a[i] -= b[i]; }
}
	xor eax,eax
	lp:
	movaps xmm0, a[eax]
	movaps xmm1, b[eax]
	movaps xmm2, xmm0
	// compare a and b values
	cmpgtps xmm0, xmm1
	// xmm3 - will hold -b
	movaps xmm3, [SIGN_BIT_MASK]
	xorps xmm3, xmm1
	// select values for the add operation,
	// true condition produce a+b, false will become a+(-b)
	// blend mask is xmm0
	blendvps xmm1,xmm3, xmm0
	addps xmm2, xmm1
	movaps a[eax], xmm2
	add eax, 16
	cmp eax, CNT
	jnz lp

Example 6-31(b) depicts an assembly sequence that uses BLENDVPS to vectorize the handling of heterogeneous computations occurring across four consecutive loop iterations.

6.6.17 VECTORIZATION OF CONTROL FLOWS IN NESTED LOOPS

The PTEST and BLENDVPx instructions can be used as building blocks to vectorize more complex control-flow statements, where each control flow statement is creating a “working” mask used as a predicate of which the conditional code under the mask will operate.

The Mandelbrot-set map evaluation is useful to illustrate a situation with more complex control flows in nested loops. The Mandelbrot-set is a set of height values mapped to a 2-D grid. The height value is the number of Mandelbrot iterations (defined over the complex number space as $I_n = I_{n-1}^2 + I_0$) needed to get $|I_n| > 2$. It is common to limit the map generation by setting some maximum threshold value of the height, all other points are assigned with a height equal to the threshold. Example 6-32 shows an example of Mandelbrot map evaluation implemented in C.

Example 6-32. Baseline C Code for Mandelbrot Set Map Evaluation

```
#define DIMX (64)
#define DIMY (64)
#define X_STEP (0.5f/DIMX)
#define Y_STEP (0.4f/(DIMY/2))
int map[DIMX][DIMY];

void mandelbrot_C()
```

Example 6-32. Baseline C Code for Mandelbrot Set Map Evaluation (Contd.)

```
{    int i,j;
     float x,y;
     for (i=0,x=-1.8f;i<DIMX;i++,x+=X_STEP)
     {
for (j=0,y=-0.2f;j<DIMY/2;j++,y+=Y_STEP)
          {float sx,sy;
               int iter = 0;
               sx = x;
               sy = y;
               while (iter < 256)
               {     if (sx*sx + sy*sy >= 4.0f)     break;
                     float old_sx = sx;
                     sx = x + sx*sx - sy*sy;
                     sy = y + 2*old_sx*sy;
                     iter++;
               }
               map[i][j] = iter;
          }
     }
}
```

Example 6-33 shows a vectorized implementation of Mandelbrot map evaluation. Vectorization is not done on the inner most loop, because the presence of the break statement implies the iteration count will vary from one pixel to the next. The vectorized version take into account the parallel nature of 2-D, vectorize over four iterations of Y values of 4 consecutive pixels, and conditionally handles three scenarios:

- In the inner most iteration, when all 4 pixels do not reach break condition, vectorize 4 pixels.
- When one or more pixels reached break condition, use blend intrinsics to accumulate the complex height vector for the remaining pixels not reaching the break condition and continue the inner iteration of the complex height vector.
- When all four pixels reached break condition, exit the inner loop.

Example 6-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics

```
__declspec(align(16)) float _INIT_Y_4[4] = {0,Y_STEP,2*Y_STEP,3*Y_STEP};
F32vec4 _F_STEP_Y(4*Y_STEP);
I32vec4 _I_ONE_ = _mm_set1_epi32(1);
F32vec4 _F_FOUR_(4.0f);
F32vec4 _F_TWO_(2.0f);;
void mandelbrot_C()
{    int i,j;
     F32vec4 x,y;

     for (i = 0, x = F32vec4(-1.8f); i < DIMX; i ++, x += F32vec4(X_STEP))
     {
for (j = DIMY/2, y = F32vec4(-0.2f) +
               *(F32vec4*)_INIT_Y_4; j < DIMY; j += 4, y += _F_STEP_Y)
```

Example 6-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics (Contd.)

```
                {     F32vec4 sx,sy;
            I32vec4 iter = _mm_setzero_si128();
            int scalar_iter = 0;
            sx = x;
            sy = y;

            while (scalar_iter < 256)
            {     int mask = 0;
                F32vec4 old_sx = sx;
                __m128 vmask = _mm_cmpnlt_ps(sx*sx + sy*sy,_F_FOUR_);
                // if all data points in our vector are hitting the “exit” condition,
                // the vectorized loop can exit
                if (_mm_test_all_ones(_mm_castps_si128(vmask)))
                    break;
                                (continue)
// if non of the data points are out, we don’t need the extra code which blends the results
                if (_mm_test_all_zeros(_mm_castps_si128(vmask),
                    _mm_castps_si128(vmask)))
                {     sx = x + sx*sx - sy*sy;
                    sy = y + _F_TWO_*old_sx*sy;
                    iter +=  _I_ONE_;
                }
                else
                {
// Blended flavour of the code, this code blends values from previous iteration with the values
// from current iteration. Only values which did not hit the “exit” condition are being stored;
// values which are already “out” are maintaining their value
                    sx = _mm_blendv_ps(x + sx*sx - sy*sy,sx,vmask);
                    sy = _mm_blendv_ps(y + _F_TWO_*old_sx*sy,sy,vmask);
                    iter = I32vec4(_mm_blendv_epi8(iter + _I_ONE_,
                        iter,_mm_castps_si128(vmask)));
                }
                scalar_iter++;
            }
            _mm_storeu_si128((__m128i*)&map[i][j],iter);
        }
    }
}
```

6.7 MEMORY OPTIMIZATIONS

You can improve memory access using the following techniques:

- Avoiding partial memory accesses.
- Increasing the bandwidth of memory fills and video fills.
- Prefetching data with Streaming SIMD Extensions. See Chapter 9.

MMX registers and XMM registers allow you to move large quantities of data without stalling the processor. Instead of loading single array values that are 8, 16, or 32 bits long, consider loading the values in a single quadword or double quadword and then incrementing the structure or array pointer accordingly.

Any data that will be manipulated by SIMD integer instructions should be loaded using either:

- An SIMD integer instruction that loads a 64-bit or 128-bit operand (for example: MOVQ MM0, M64).
- The register-memory form of any SIMD integer instruction that operates on a quadword or double quadword memory operand (for example, PMADDW MM0, M64).

All SIMD data should be stored using an SIMD integer instruction that stores a 64-bit or 128-bit operand (for example: MOVQ M64, MM0).

The goal of the above recommendations is twofold. First, the loading and storing of SIMD data is more efficient using the larger block sizes. Second, following the above recommendations helps to avoid mixing of 8-, 16-, or 32-bit load and store operations with SIMD integer technology load and store operations to the same SIMD data.

This prevents situations in which small loads follow large stores to the same area of memory, or large loads follow small stores to the same area of memory.

6.7.1 PARTIAL MEMORY ACCESSES

Consider a case with a large load after a series of small stores to the same area of memory (beginning at memory address MEM). The large load stalls in the case shown in Example 6-34.

Example 6-34. A Large Load after a Series of Small Stores (Penalty)

```
mov    mem, eax        ; store dword to address “mem"
mov    mem + 4, ebx    ; store dword to address “mem + 4"
   :
   :
movq   mm0, mem        ; load qword at address “mem", stalls
```

MOVQ must wait for the stores to write memory before it can access all data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory). When you change the code sequence as shown in Example 6-35, the processor can access the data without delay.

Example 6-35. Accessing Data without Delay

```
movd   mm1, ebx        ; build data into a qword first
                       ; before storing it to memory
movd   mm2, eax
psllq  mm1, 32
```

Example 6-35. Accessing Data without Delay (Contd.)

```
por     mm1, mm2
movq    mem, mm1        ; store SIMD variable to “mem" as
                        ; a qword
    :
    :
movq    mm0, mem        ; load qword SIMD “mem", no stall
```

Consider a case with a series of small loads after a large store to the same area of memory (beginning at memory address MEM), as shown in Example 6-36. Most of the small loads stall because they are not aligned with the store. See Section 3.6.4 for details.

Example 6-36. A Series of Small Loads after a Large Store

```
movq    mem, mm0        ; store qword to address “mem"
    :
    :
mov     bx, mem + 2     ; load word at “mem + 2" stalls
mov     cx, mem + 4     ; load word at “mem + 4" stalls
```

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example: when doublewords or words are stored and then words or bytes are read from the same area of memory).

When you change the code sequence as shown in Example 6-37, the processor can access the data without delay.

Example 6-37. Eliminating Delay for a Series of Small Loads after a Large Store

```
movq    mem, mm0    ; store qword to address “mem"
    :
    :

movq    mm1, mem    ; load qword at address “mem"
movd    eax, mm1    ; transfer “mem + 2" to eax from
                    ; MMX register, not memory

psrlq   mm1, 32
shr     eax, 16
movd    ebx, mm1    ; transfer “mem + 4" to bx from
                    ; MMX register, not memory
and     ebx, 0ffffh
```

These transformations, in general, increase the number of instructions required to perform the desired operation.

6.7.2 INCREASING BANDWIDTH OF MEMORY FILLS AND VIDEO FILLS

It is beneficial to understand how memory is accessed and filled. A memory-to-memory fill (for example a memory-to-video fill) is defined as a 64-byte (cache line) load from memory which is immediately stored back to memory (such as a video frame buffer).

The following are guidelines for obtaining higher bandwidth and shorter latencies for sequential memory fills (video fills). These recommendations are relevant for all Intel architecture processors with MMX technology and refer to cases in which the loads and stores do not hit in the first- or second-level cache.

6.7.2.1 Increasing Memory Bandwidth Using the MOVDQ Instruction

Loading any size data operand will cause an entire cache line to be loaded into the cache hierarchy. Thus, any size load looks more or less the same from a memory bandwidth perspective. However, using many smaller loads consumes more microarchitectural resources than fewer larger stores. Consuming too many resources can cause the processor to stall and reduce the bandwidth that the processor can request of the memory subsystem.

Using MOVDQ to store the data back to UC memory (or WC memory in some cases) instead of using 32-bit stores (for example, MOVD) will reduce by three-quarters the number of stores per memory fill cycle. As a result, using the MOVDQ in memory fill cycles can achieve significantly higher effective bandwidth than using MOVD.

6.7.2.2 Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page

DRAM is divided into pages, which are not the same as operating system (OS) pages. The size of a DRAM page is a function of the total size of the DRAM and the organization of the DRAM. Page sizes of several Kilobytes are common. Like OS pages, DRAM pages are constructed of sequential addresses. Sequential memory accesses to the same DRAM page have shorter latencies than sequential accesses to different DRAM pages.

In many systems the latency for a page miss (that is, an access to a different page instead of the page previously accessed) can be twice as large as the latency of a memory page hit (access to the same page as the previous access). Therefore, if the loads and stores of the memory fill cycle are to the same DRAM page, a significant increase in the bandwidth of the memory fill cycles can be achieved.

6.7.2.3 Increasing UC and WC Store Bandwidth by Using Aligned Stores

Using aligned stores to fill UC or WC memory will yield higher bandwidth than using unaligned stores. If a UC store or some WC stores cross a cache line boundary, a single store will result in two transaction on the bus, reducing the efficiency of the bus transactions. By aligning the stores to the size of the stores, you eliminate the possibility of crossing a cache line boundary, and the stores will not be split into separate transactions.

6.7.3 REVERSE MEMORY COPY

Copying blocks of memory from a source location to a destination location in reverse order presents a challenge for software to make the most out of the machines capabilities while avoiding microarchitectural hazards. The basic, un-optimized C code is shown in Example 6-38.

The simple C code in Example 6-38 is sub-optimal, because it loads and stores one byte at a time (even in situations that hardware prefetcher might have brought data in from system memory to cache).

Example 6-38. Un-optimized Reverse Memory Copy in C

```
unsigned char* src;
unsigned char* dst;
while (len > 0)
{
*dst-- = *src++;
--len;
}
```

Using MOVDQA or MOVDQU, software can load and store up to 16 bytes at a time but must either ensure 16 byte alignment requirement (if using MOVDQA) or minimize the delays MOVDQU may encounter if data span across cache line boundary.

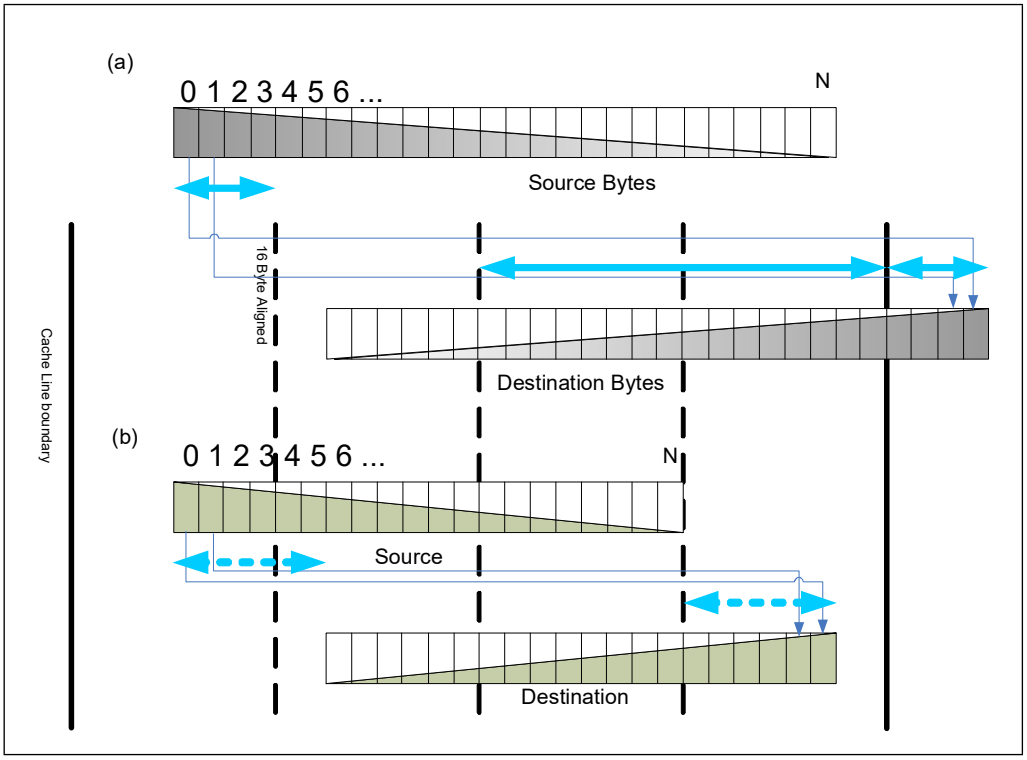


Figure 6-8. Data Alignment of Loads and Stores in Reverse Memory Copy

Given the general problem of arbitrary byte count to copy, arbitrary offsets of leading source byte and destination bytes, address alignment relative to 16 byte and cache line boundaries, these alignment situations can be a bit complicated. Figure 6-8 (a) and (b) depict the alignment situations of reverse memory copy of N bytes.

The general guidelines for dealing with unaligned loads and stores are (in order of importance):

- Avoid stores that span cache line boundaries.
- Minimize the number of loads that span cacheline boundaries.
- Favor 16-byte aligned loads and stores over unaligned versions.

In Figure 6-8 (a), the guidelines above can be applied to the reverse memory copy problem as follows:

1. Peel off several leading destination bytes until it aligns on 16 Byte boundary, then the ensuing destination bytes can be written to using MOVAPS until the remaining byte count falls below 16 bytes.
2. After the leading source bytes have been peeled (corresponding to step 1 above), the source alignment in Figure 6-8 (a) allows loading 16 bytes at a time using MOVAPS until the remaining byte count falls below 16 bytes.

Switching the byte ordering of each 16 bytes of data can be accomplished by a 16-byte mask with PSHUFB. The pertinent code sequence is shown in Example 6-39.

Example 6-39. Using PSHUFB to Reverse Byte Ordering 16 Bytes at a Time

```
__declspec(align(16)) static const unsigned char BswapMask[16] = {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
        mov esi, src
        mov edi, dst
        mov ecx, len
        movaps xmm7, BswapMask
start:
        movdqa xmm0, [esi]
        pshufb xmm0, xmm7
        movdqa [edi-16], xmm0
sub edi, 16
        add esi, 16
        sub ecx, 16
        cmp ecx, 32
        jae start
        //handle left-overs
```

In Figure 6-8 (b), we also start with peeling the destination bytes:

1. Peel off several leading destination bytes until it aligns on 16 Byte boundary, then the ensuing destination bytes can be written to using MOVAPS until the remaining byte count falls below 16 bytes. However, the remaining source bytes are not aligned on 16 byte boundaries, replacing MOVDQA with MOVDQU for loads will inevitably run into cache line splits.
2. To achieve higher data throughput than loading unaligned bytes with MOVDQU, the 16 bytes of data targeted to each of 16 bytes of aligned destination addresses can be assembled using two aligned loads. This technique is illustrated in Figure 6-9.

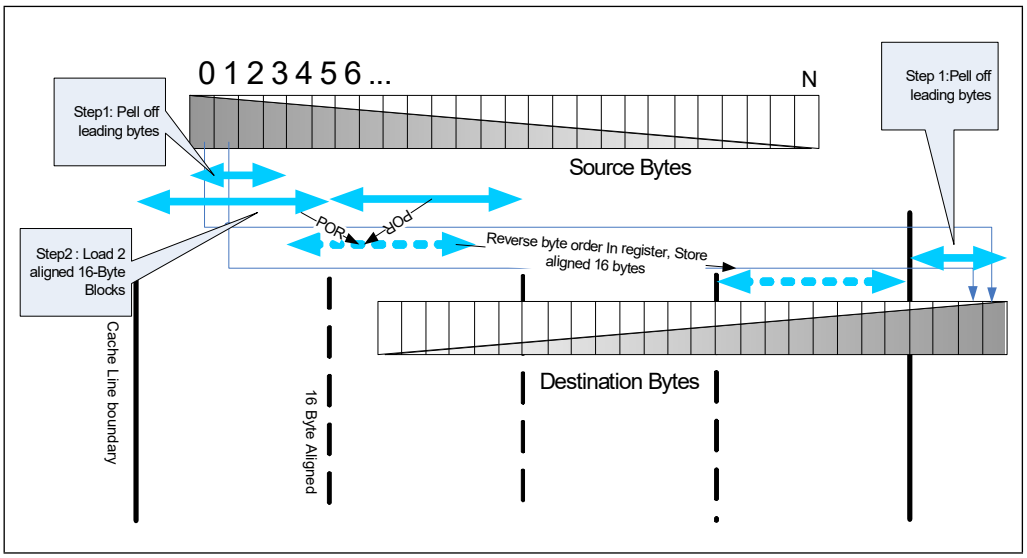


Figure 6-9. A Technique to Avoid Cacheline Split Loads in Reverse Memory Copy Using Two Aligned Loads

6.8 CONVERTING FROM 64-BIT TO 128-BIT SIMD INTEGERS

SSE2 defines a superset of 128-bit integer instructions currently available in MMX technology; the operation of the extended instructions remains. The superset simply operates on data that is twice as wide. This simplifies porting of 64-bit integer applications. However, there are few considerations:

- Computation instructions which use a memory operand that may not be aligned to a 16-byte boundary must be replaced with an unaligned 128-bit load (MOVDQU) followed by the same computation operation that uses instead register operands.

  Use of 128-bit integer computation instructions with memory operands that are not 16-byte aligned will result in a #GP. Unaligned 128-bit loads and stores are not as efficient as corresponding aligned versions; this fact can reduce the performance gains when using the 128-bit SIMD integer extensions.

- General guidelines on the alignment of memory operands are:
  - The greatest performance gains can be achieved when all memory streams are 16-byte aligned.
  - Reasonable performance gains are possible if roughly half of all memory streams are 16-byte aligned and the other half are not.
  - Little or no performance gain may result if all memory streams are not aligned to 16-bytes. In this case, use of the 64-bit SIMD integer instructions may be preferable.
- Loop counters need to be updated because each 128-bit integer instruction operates on twice the amount of data as its 64-bit integer counterpart.
- Extension of the PSHUFW instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: PSHUFHW, PSHUFLW, and PSHUFD.
- Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) are extended to 128 bits by:
  - Use of PSRLQ and PSLLQ, along with masking logic operations.
  - A Code sequence rewritten to use the PSRLDQ and PSLLDQ instructions (shift double quad-word operand by bytes).

6.8.1 SIMD OPTIMIZATIONS AND MICROARCHITECTURES

Intel Core processors have a different microarchitecture than Intel NetBurst microarchitecture. The following sections discuss optimizing SIMD code that targets Intel Core Solo and Intel Core Duo processors.

On Intel Core Solo and Intel Core Duo processors, LDDQU behaves identically to movdqu by loading 16 bytes of data irrespective of address alignment.

6.8.1.1 Packed SSE2 Integer versus MMX Instructions

In general, 128-bit SIMD integer instructions should be favored over 64-bit MMX instructions on Intel Core Solo and Intel Core Duo processors. This is because:

- Improved decoder bandwidth and more efficient micro-op flows.
- Wider width of the XMM registers can benefit code that is limited by either decoder bandwidth or execution latency. XMM registers can provide twice the space to store data for in-flight execution. Wider XMM registers can facilitate loop-unrolling or in reducing loop overhead by halving the number of loop iterations.

In microarchitectures prior to Intel Core microarchitecture, execution throughput of 128-bit SIMD integration operations is basically the same as 64-bit MMX operations. Some shuffle/unpack/shift operations do not benefit from the front end improvements. The net impact of using 128-bit SIMD integer instruction on Intel Core Solo and Intel Core Duo processors is likely to be slightly positive overall, but there may be a few situations where their use will generate an unfavorable performance impact.

Intel Core microarchitecture generally executes 128-bit SIMD instructions more efficiently than previous microarchitectures in terms of latency and throughput, many of the limitations specific to Intel Core Duo, Intel Core Solo processors do not apply. The same is true of Intel Core microarchitecture relative to Intel NetBurst microarchitectures.

Enhanced Intel Core microarchitecture provides even more powerful 128-bit SIMD execution capabilities and more comprehensive sets of SIMD instruction extensions than Intel Core microarchitecture. The integer SIMD instructions offered by SSE4.1 operates on 128-bit XMM register only. All of these highly encourages software to favor 128-bit vectorizable code to take advantage of processors based on Enhanced Intel Core microarchitecture and Intel Core microarchitecture.

6.9 TUNING PARTIALLY VECTORIZABLE CODE

Some loop structured code are more difficult to vectorize than others. Example 6-40 depicts a loop carrying out table look-up operation and some arithmetic computation.

Example 6-40. Table Look-up Operations in C Code

```
// pIn1        integer input arrays.
// pOut        integer output array.
// count       size of array.
// LookUpTable  integer values.
TABLE_SIZE        size of the look-up table.
for (unsigned i=0; i < count; i++)
{    pOut[i] =
     ( ( LookUpTable[pIn1[i] % TABLE_SIZE] + pIn1[i] + 17 ) | 17
     ) % 256;
}
```

Although some of the arithmetic computations and input/output to data array in each iteration can be easily vectorizable, but the table look-up via an index array is not. This creates different approaches to tuning. A compiler can take a scalar approach to execute each iteration sequentially. Hand-tuning of such loops may use a couple of different techniques to handle the non-vectorizable table look-up operation. One vectorization technique is to load the input data for four iteration at once, then use SSE2 instruction to shift out individual index out of an XMM register to carry out table look-up sequentially. The shift technique is depicted by Example 6-41. Another technique is to use PEXTRD in SSE4.1 to extract the index from an XMM directly and then carry out table look-up sequentially. The PEXTRD technique is depicted by Example 6-42.

Example 6-41. Shift Techniques on Non-Vectorizable Table Look-up

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
    mov     esi, pIn1
    mov     ebx, pOut
    mov     ecx, count
    mov     edx, pLookUpTablePTR
    movaps  xmm6, modulo
    movaps  xmm5, c
lloop:
// vectorizable multiple consecutive data accesses
    movaps      xmm4, [esi]         // read 4 indices from pIn1
    movaps      xmm7, xmm4
    pand        xmm7, tableSize
//Table look-up is not vectorizable, shift out one data element to look up table one by one
    movd        eax, xmm7           // get first index
    movd        xmm0, word ptr[edx + eax*4]
    psrldq      xmm7, 4
    movd        eax, xmm7           // get 2nd index
    movd        xmm1, word ptr[edx + eax*4]
    psrldq      xmm7, 4
    movd        eax, xmm7           // get 3rdindex
    movd        xmm2, word ptr[edx + eax*4]
```

Example 6-41. Shift Techniques on Non-Vectorizable Table Look-up (Contd.)

```
    psrldq      xmm7, 4
    movd        eax, xmm7       // get fourth index
    movd        xmm3, word ptr[edx + eax*4]
    movd        xmm2, word ptr[edx + eax*4]
    psrldq      xmm7, 4
    movd        eax, xmm7       // get fourth index
    movd        xmm3, word ptr[edx + eax*4]
//end of scalar part
//packing
    movlhps     xmm1,xmm3
    psllq       xmm1,32
    movlhps     xmm0,xmm2
    orps        xmm0,xmm1
//end of packing
//Vectorizable computation operations
    paddd       xmm0, xmm4 //+pIn1
    paddd       xmm0, xmm5 // +17
    por         xmm0, xmm5
    andps       xmm0, xmm6 //mod
    movaps      [ebx], xmm0
//end of vectorizable operation

    add         ebx, 16
    add         esi, 16
    add         edi, 16
    sub         ecx, 1
    test        ecx, ecx
    jne lloop
```

Example 6-42. PEXTRD Techniques on Non-Vectorizable Table Look-up

```
int modulo[4] = {256-1, 256-1, 256-1, 256-1};
int c[4] = {17, 17, 17, 17};
    mov     esi, pIn1
    mov     ebx, pOut
    mov     ecx, count
    mov     edx, pLookUpTablePTR
    movaps  xmm6, modulo
    movaps  xmm5, c
lloop:
// vectorizable multiple consecutive data accesses
    movaps      xmm4, [esi]     // read 4 indices from pIn1
    movaps      xmm7, xmm4
    pand        xmm7, tableSize
//Table look-up is not vectorizable, extract one data element to look up table one by one
    movd        eax, xmm7       // get first index
    mov         eax, [edx + eax*4]
    movd        xmm0, eax
```

Example 6-42. PEXTRD Techniques on Non-Vectorizable Table Look-up (Contd.)

```
    pextrd      eax, xmm7, 1      // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 1
    pextrd      eax, xmm7, 2      // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 2
    pextrd      eax, xmm7, 3      // extract 2nd index
    mov         eax, [edx + eax*4]
    pinsrd      xmm0, eax, 2
//end of scalar part
//packing not needed
//Vectorizable operations
    paddd       xmm0, xmm4 //+pIn1
    paddd       xmm0, xmm5 // +17
    por         xmm0, xmm5
    andps       xmm0, xmm6 //mod
    movaps      [ebx], xmm0
    add         ebx, 16
    add         esi, 16
    add         edi, 16
    sub         ecx, 1
    test        ecx, ecx
    jne lloop
```

The effectiveness of these two hand-tuning techniques on partially vectorizable code depends on the relative cost of transforming data layout format using various forms of pack and unpack instructions.

The shift technique requires additional instructions to pack scalar table values into an XMM to transition into vectorized arithmetic computations. The net performance gain or loss of this technique will vary with the characteristics of different microarchitectures. The alternate PEXTRD technique uses less instruction to extract each index, does not require extraneous packing of scalar data into packed SIMD data format to begin vectorized arithmetic computation.

6.10 PARALLEL MODE AES ENCRYPTION AND DECRYPTION

To deliver optimal encryption and decryption throughput using AESNI, software can optimize by re-ordering the computations and working on multiple blocks in parallel. This can speed up encryption (and decryption) in parallel modes of operation such as ECB, CTR, and CBC-Decrypt (comparing to CBC-Encrypt which is serial mode of operation). See details in Recommendation for Block Cipher Modes of Operation?. The Related Documentation section provides a pointer to this document.

In Sandy Bridge microarchitecture, the AES round instructions (AESENC / AESECNLAST / AESDEC / AESDECLAST) have a throughput of one cycle and latency of eight cycles. This allows independent AES instructions for multiple blocks to be dispatched every cycle, if data can be provided sufficiently fast. Compared to the prior Westmere microarchitecture, where these instructions have throughput of two cycles and a latency of six cycles, the AES encryption/decryption throughput can be significantly increased for parallel modes of operation.

To achieve optimal parallel operation with multiple blocks, write the AES software sequences in a way that it computes one AES round on multiple blocks, using one Round Key, and then it continues to compute the subsequent round for multiple blocks, using another Round Key.

For such software optimization, you need to define the number of blocks that are processed in parallel. In Sandy Bridge microarchitecture, the optimal parallelization parameter is eight blocks, compared to four blocks on prior microarchitecture.

6.10.1 AES COUNTER MODE OF OPERATION

Example 6-43 is an example of a function that implements the Counter Mode (CTR mode) of operations while operating on eight blocks in parallel. The following pseudo-code encrypts n data blocks of 16 byte each (PT[i]):

Example 6-43. Pseudo-Code Flow of AES Counter Mode Operation

```
CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
    CT[i] := PT[i] XOR AES(CTRBLK)
    CTRBLK := CTRBLK + 1) % 256;
END
CT[n] := PT[n] XOR TRUNC(AES(CTRBLK)) CTRBLK := NONCE || IV || ONE
FOR i := 1 to n-1 DO
    CT[i] := PT[i] XOR AES(CTRBLK)// CT [i] is the i-th ciphetext block
    CTRBLK := CTRBLK + 1
END
CT[n]:= PT[n] XOR TRUNC(AES(CTRBLK))
```

Example 6-44 in the following pages show the assembly implementation of the above code, optimized for Sandy Bridge microarchitecture.

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel

```
/*****************************************************************************/
/* This function encrypts an input buffer using AES in CTR mode            */
/* The parameters:                                                          */
/*   const unsigned char *in - pointer to the palintext for encryption or   */
/* ciphertextfor decryption                */
/*   unsigned char *out - pointer to the buffer where the encrypted/decrypted*/
/*                  data will be stored                                     */
/*   const unsigned char ivec[8] - 8 bytes of the initialization vector     */
/*   const unsigned char nonce[4] - 4 bytes of the nonce                    */
/*   const unsigned long length - the length of the input in bytes          */
/*   int number_of_rounds - number of AES round. 10 = AES128, 12 = AES192, 14 = AES256 */
/*   unsigned char *key_schedule - pointer to the AES key schedule          */
/*****************************************************************************/
//void AES_128_CTR_encrypt_parallelize_8_blocks_unrolled (
//                              const unsigned char *in,
//                              unsigned char *out,
//                              const unsigned char ivec[8],
//                              const unsigned char nonce[4],
//                              const unsigned long length,
//                              unsigned char *key_schedule)
.align   16,0x90
.align   16

ONE:              .quad 0x00000000,0x00000001
.align   16
FOUR:             .quad 0x00000004,0x00000004
.align   16
EIGHT:            .quad 0x00000008,0x00000008

.align   16
TWO_N_ONE:    .quad 0x00000002,0x00000001
.align   16
TWO_N_TWO:    .quad 0x00000002,0x00000002
.align   16
LOAD_HIGH_BROADCAST_AND_BSWAP:    .byte 15,14,13,12,11,10,9,8
                                  .byte 15,14,13,12,11,10,9,8
align   16
BSWAP_EPI_64:             .byte 7,6,5,4,3,2,1,0
                          .byte 15,14,13,12,11,10,9,8
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
.globl AES_CTR_encrypt

AES_CTR_encrypt:
# parameter 1: %rdi         # parameter 2: %rsi
# parameter 3: %rdx         # parameter 4: %rcx
# parameter 5: %r8          # parameter 6: %r9
# parameter 7: 8 + %rsp
movq      %r8, %r10
      movl      8(%rsp), %r12d
      shrq      $4, %r8
      shlq      $60, %r10
      je        NO_PARTS
      addq      $1, %r8
NO_PARTS:
      movq      %r8, %r10
      shlq      $61, %r10
      shrq      $61, %r10

      pinsrq    $1, (%rdx), %xmm0
      pinsrd    $1, (%rcx), %xmm0
      psrldq    $4, %xmm0
      movdqa    %xmm0, %xmm4
      pshufb    (LOAD_HIGH_BROADCAST_AND_BSWAP), %xmm4
      paddq     (TWO_N_ONE), %xmm4
      movdqa    %xmm4, %xmm1
      paddq     (TWO_N_TWO), %xmm4
      movdqa    %xmm4, %xmm2
      paddq     (TWO_N_TWO), %xmm4
      movdqa    %xmm4, %xmm3
     paddq      (TWO_N_TWO), %xmm4
     pshufb     (BSWAP_EPI_64), %xmm1
    paddq       (TWO_N_TWO), %xmm4
    pshufb      (BSWAP_EPI_64), %xmm1
    pshufb      (BSWAP_EPI_64), %xmm2
    pshufb      (BSWAP_EPI_64), %xmm3
    pshufb      (BSWAP_EPI_64), %xmm4
shrq      $3, %r8
      je        REMAINDER
      subq      $128, %rsi
      subq      $128, %rdi
LOOP:
      addq      $128, %rsi
      addq      $128, %rdi
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    movdqa    %xmm0, %xmm7
    movdqa    %xmm0, %xmm8
    movdqa    %xmm0, %xmm9
    movdqa    %xmm0, %xmm10
    movdqa    %xmm0, %xmm11
    movdqa    %xmm0, %xmm12
    movdqa    %xmm0, %xmm13
    movdqa    %xmm0, %xmm14

    shufpd    $2, %xmm1, %xmm7
    shufpd    $0, %xmm1, %xmm8
    shufpd    $2, %xmm2, %xmm9
    shufpd    $0, %xmm2, %xmm10
    shufpd    $2, %xmm3, %xmm11
    shufpd    $0, %xmm3, %xmm12
    shufpd    $2, %xmm4, %xmm13
    shufpd    $0, %xmm4, %xmm14

    pshufb    (BSWAP_EPI_64), %xmm1
    pshufb    (BSWAP_EPI_64), %xmm2
    pshufb    (BSWAP_EPI_64), %xmm3
    pshufb    (BSWAP_EPI_64), %xmm4

    movdqa    (%r9), %xmm5
    movdqa    16(%r9), %xmm6

    paddq     (EIGHT), %xmm1
    paddq     (EIGHT), %xmm2
    paddq     (EIGHT), %xmm3
    paddq     (EIGHT), %xmm4

    pxor      %xmm5, %xmm7
    pxor      %xmm5, %xmm8
    pxor      %xmm5, %xmm9
    pxor      %xmm5, %xmm10
    pxor      %xmm5, %xmm11
    pxor      %xmm5, %xmm12
    pxor      %xmm5, %xmm13
    pxor      %xmm5, %xmm14

    pshufb    (BSWAP_EPI_64), %xmm1
    pshufb    (BSWAP_EPI_64), %xmm2
    pshufb    (BSWAP_EPI_64), %xmm3
    pshufb    (BSWAP_EPI_64), %xmm4
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14

movdqa   32(%r9), %xmm5
movdqa   48(%r9), %xmm6

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14

movdqa   64(%r9), %xmm5
movdqa   80(%r9), %xmm6
 aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11


aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
movdqa   96(%r9), %xmm5
movdqa   112(%r9), %xmm6

aesenc   %xmm5, %xmm7
aesenc   %xmm5, %xmm8
aesenc   %xmm5, %xmm9
aesenc   %xmm5, %xmm10
aesenc   %xmm5, %xmm11
aesenc   %xmm5, %xmm12
aesenc   %xmm5, %xmm13
aesenc   %xmm5, %xmm14

aesenc   %xmm6, %xmm7
aesenc   %xmm6, %xmm8
aesenc   %xmm6, %xmm9
aesenc   %xmm6, %xmm10
aesenc   %xmm6, %xmm11
aesenc   %xmm6, %xmm12
aesenc   %xmm6, %xmm13
aesenc   %xmm6, %xmm14


 movdqa   128(%r9), %xmm5
 movdqa   144(%r9), %xmm6
 movdqa   160(%r9), %xmm15
 cmp      $12, %r12d

 aesenc   %xmm5, %xmm7
 aesenc   %xmm5, %xmm8
 aesenc   %xmm5, %xmm9
 aesenc   %xmm5, %xmm10
 aesenc   %xmm5, %xmm11
 aesenc   %xmm5, %xmm12
 aesenc   %xmm5, %xmm13
 aesenc   %xmm5, %xmm14

 aesenc   %xmm6, %xmm7
 aesenc   %xmm6, %xmm8
 aesenc   %xmm6, %xmm9
 aesenc   %xmm6, %xmm10
 aesenc   %xmm6, %xmm11
 aesenc   %xmm6, %xmm12
 aesenc   %xmm6, %xmm13
 aesenc   %xmm6, %xmm14

 jb       LAST
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    movdqa    160(%r9), %xmm5
    movdqa    176(%r9), %xmm6
    movdqa    192(%r9), %xmm15
    cmp       $14, %r12d

    aesenc    %xmm5, %xmm7
    aesenc    %xmm5, %xmm8
    aesenc    %xmm5, %xmm9
    aesenc    %xmm5, %xmm10
    aesenc    %xmm5, %xmm11
    aesenc    %xmm5, %xmm12
    aesenc    %xmm5, %xmm13
    aesenc    %xmm5, %xmm14

    jb        LAST

    movdqa    192(%r9), %xmm5
    movdqa    208(%r9), %xmm6
    movdqa    224(%r9), %xmm15

    aesenc    %xmm5, %xmm7

    aesenc    %xmm5, %xmm8
    aesenc    %xmm5, %xmm9
    aesenc    %xmm5, %xmm10
    aesenc    %xmm5, %xmm11
    aesenc    %xmm5, %xmm12
    aesenc    %xmm5, %xmm13
    aesenc    %xmm5, %xmm14

    aesenc    %xmm6, %xmm7
    aesenc    %xmm6, %xmm8
    aesenc    %xmm6, %xmm9
    aesenc    %xmm6, %xmm10
    aesenc    %xmm6, %xmm11
    aesenc    %xmm6, %xmm12
    aesenc    %xmm6, %xmm13
    aesenc    %xmm6, %xmm14
LAST:
    aesenc    %xmm5, %xmm12
    aesenc    %xmm5, %xmm13
    aesenc    %xmm5, %xmm14
    aesenclast %xmm15, %xmm7
    aesenclast %xmm15, %xmm8
    aesenclast %xmm15, %xmm9
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    aesenclast %xmm15, %xmm10
    aesenclast %xmm15, %xmm11
    aesenclast %xmm15, %xmm12
    aesenclast %xmm15, %xmm13
    aesenclast %xmm15, %xmm14

    pxor    (%rdi), %xmm7
    pxor    16(%rdi), %xmm8
    pxor    32(%rdi), %xmm9
    pxor    48(%rdi), %xmm10
    pxor    64(%rdi), %xmm11
    pxor    80(%rdi), %xmm12
    pxor    96(%rdi), %xmm13
    pxor    112(%rdi), %xmm14

    dec %r8

    movdqu  %xmm7, (%rsi)
    movdqu  %xmm8, 16(%rsi)
    movdqu  %xmm9, 32(%rsi)
    movdqu  %xmm10, 48(%rsi)
    movdqu  %xmm11, 64(%rsi)
    movdqu  %xmm12, 80(%rsi)
    movdqu  %xmm13, 96(%rsi)
    movdqu  %xmm14, 112(%rsi)
jne LOOP
addq $128,%rsi
    addq $128,%rdi

REMAINDER:
      cmp $0, %r10
      je END
      shufpd  $2, %xmm1, %xmm0
IN_LOOP:
      movdqa %xmm0, %xmm11
      pshufb (BSWAP_EPI_64), %xmm0
      pxor   (%r9), %xmm11
      paddq  (ONE), %xmm0
      aesenc 16(%r9), %xmm11
      aesenc 32(%r9), %xmm11
      pshufb (BSWAP_EPI_64), %xmm0
      aesenc 48(%r9), %xmm11
      aesenc 64(%r9), %xmm11
      aesenc 80(%r9), %xmm11
      aesenc 96(%r9), %xmm11
      aesenc 112(%r9), %xmm11
      aesenc 128(%r9), %xmm11
```

Example 6-44. AES128-CTR Implementation with Eight Block in Parallel (Contd.)

```
    aesenc 48(%r9), %xmm11
    aesenc 64(%r9), %xmm11
    aesenc 80(%r9), %xmm11
    aesenc 96(%r9), %xmm11
    aesenc 112(%r9), %xmm11
    aesenc 128(%r9), %xmm11

    jb    IN_LAST

    aesenc 160(%r9), %xmm11
    aesenc 176(%r9), %xmm11
    movdqa 192(%r9), %xmm2
    cmp    $14, %r12d

    jb    IN_LAST

    aesenc 192(%r9), %xmm11
    aesenc 208(%r9), %xmm11
    movdqa 224(%r9), %xmm2

IN_LAST:
aesenclast %xmm2, %xmm11
    pxor      (%rdi) ,%xmm11
    movdqu    %xmm11, (%rsi)
    addq      $16,%rdi
    addq      $16,%rsi
    dec       %r10
    jne       IN_LOOP
END:
    ret
```

6.10.2 AES KEY EXPANSION ALTERNATIVE

In Sandy Bridge microarchitecture, the throughput of AESKEYGENASSIST is two cycles with higher latency than the AESENC/AESDEC instructions. Software may consider implementing the AES key expansion by using the AESENCLAST instruction with the second operand (i.e., the round key) being the RCON value, duplicated four times in the register. The AESENCLAST instruction performs the SubBytes step and the xor-with-RCON step, while the ROTWORD step can be done using a PSHUFB instruction. Following are code examples of AES128 key expansion using either method.

Example 6-45. AES128 Key Expansion

Use AESKENYGENASSIST	Use AESENCLAST

.align 16,0x90
.globl AES_128_Key_Expansion
AES_128_Key_Expansion:
# parameter 1: %rdi 	
# parameter 2: %rsi 	
movl $10, 240(%rsi)
movdqu (%rdi), %xmm1
movdqa %xmm1, (%rsi)
 	
aeskeygenassist $1, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 16(%rsi)
aeskeygenassist $2, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 32(%rsi)
aeskeygenassist $4, %xmm1, %xmm2
 	
ASSISTS:
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 48(%rsi)
aeskeygenassist $8, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 64(%rsi)
aeskeygenassist $16, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 80(%rsi)
aeskeygenassist $32, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 96(%rsi)
aeskeygenassist $64, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 112(%rsi)
aeskeygenassist $0x80, %xmm1, %xmm2
call PREPARE_ROUNDKEY_128
movdqa %xmm1, 128(%rsi)
	mask:
	.long
	0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d
	con1:
	.long 1,1,1,1
	con2:
	.long 0x1b,0x1b,0x1b,0x1b
	.align 16,0x90
	.globl AES_128_Key_Expansion
	
	AES_128_Key_Expansion:
	# parameter 1: %rdi
	# parameter 2: %rsi
	movdqu (%rdi), %xmm1
	movdqa %xmm1, (%rsi)
	movdqa %xmm1, %xmm2
	movdqa (con1), %xmm0
	movdqa (mask), %xmm15
	mov $8, %ax
	LOOP1:
	add $16, %rsi
	dec %ax
	pshufb %xmm15,%xmm2
	aesenclast %xmm0, %xmm2
	pslld $1, %xmm0
	movdqa %xmm1, %xmm3
	pslldq $4, %xmm3
	pxor %xmm3, %xmm1
	pslldq $4, %xmm3
	pxor %xmm3, %xmm1
	pslldq $4, %xmm3
	pxor %xmm3, %xmm1
	pxor %xmm2, %xmm1
	movdqa %xmm1, (%rsi)
	movdqa %xmm1, %xmm2
	jne LOOP1

aeskeygenassist $0x1b, %xmm1, %xmm2 call PREPARE_ROUNDKEY_128 movdqa %xmm1, 144(%rsi) aeskeygenassist $0x36, %xmm1, %xmm2 call PREPARE_ROUNDKEY_128 movdqa %xmm1, 160(%rsi) ret	movdqa (con2), %xmm0 pshufb %xmm15,%xmm2 aesenclast %xmm0, %xmm2 pslld $1, %xmm0 movdqa %xmm1, %xmm3 pslldq $4, %xmm3 pxor %xmm3, %xmm1

Example 6-45. AES128 Key Expansion (Contd.)

Use AESKENYGENASSIST	Use AESENCLAST
PREPARE_ROUNDKEY_128: pshufd $255, %xmm2, %xmm2 movdqa %xmm1, %xmm3 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 ret	pslldq $4, %xmm3 pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 movdqa %xmm1, 16(%rsi) movdqa %xmm1, %xmm2 pshufb %xmm15,%xmm2 aesenclast %xmm0, %xmm2 movdqa %xmm1, %xmm3 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pslldq $4, %xmm3 pxor %xmm3, %xmm1 pxor %xmm2, %xmm1 movdqa %xmm1, 32(%rsi) movdqa %xmm1, %xmm2 ret

6.10.3 ENHANCEMENT IN HASWELL MICROARCHITECTURE

6.10.3.1 Advanced Encryption Standard (AES) and Multi-Buffer Cryptographic Throughput

The AESINC/AESINCLAST, AESDEC/AESDECLAST instructions in the Haswell microarchitecture have slightly improved latency, and are one micro-op. These improvements are expected to benefit AES algorithms operating in parallel modes (e.g., CBC decryption) and multiple-buffer implementations of AES algorithms. See the following link for additional details on AESNI:

- Intel® Advanced Encryption Standard (Intel® AES) Instructions Set - Rev 3.01

6.11 LIGHT-WEIGHT DECOMPRESSION AND DATABASE PROCESSING

Traditionally, database storage requires high-compression ratio means to preserve the finite disk I/O bandwidth limitations. In row-optimized database architecture, the primary limitation on database processing performance often correlates to the hardware constraints of the storage I/O bandwidth, the locality issues of data records from rows in large tables that must be decompressed from its storage format. Many recent database innovations are centered around columnar database architecture, where storage format is optimized for query operations to fetch data in a sequential manner.

Some of the recent advances in columnar database (also known as in-memory database) are light-weight compression/decompression techniques and vectorized query operation primitives using SSE4.2 and other SIMD instructions. When a database engine combines those processing techniques with a column-optimized storage system using solid state drives, query performance increase of several fold has been reported[1]. This section discusses the usage of SIMD instructions for light-weight compression/decompression in columnar databases.

1. See published TPC-H non-clustered performance results at www.tpc.org

The optimal objective for light-weight compression/decompression is to deliver high throughput at reasonably low CPU utilization, so the finite total compute bandwidth can be divided more favorably between query processing and decompression to achieve maximal query throughput. SSE4.2 can raise the compute bandwidth for some query operations to a significantly higher level (see Section 14.3.3), compared to query primitives implemented using general-purpose-register instructions. This also places higher demand on the streaming data feed of decompressed columnar data.

6.11.1 REDUCED DYNAMIC RANGE DATASETS

One of the more successful approaches to compress/decompress columnar data in high-speed is based on the idea that an ensemble of integral values in a sequential data stream of fixed-size storage width can be represented more compactly if the dynamic range of that ensemble is reduced by way of partitioning, offset from a common reference value, and additional techniques[1],[2].

For example, a column that stores 5-digit ZIPCODE as 32-bit integers only requires a dynamic range of 17 bits. The unique primary keys in a 2 billion row table can be reduced through partitioning of sequential blocks of 2^N entries to store the offset in the block header and reducing the storage size of each 32-bit integer as N bits.

6.11.2 COMPRESSION AND DECOMPRESSION USING SIMD INSTRUCTIONS

To illustrate the usage of SIMD instructions for reduced-dynamic-range compression/decompression, and compressed data elements are not byte-aligned, we consider an array of 32-bit integers whose dynamic range only requires 5 bits per value.

To pack a stream of 32-bit integer values into consecutive 5-bit buckets, the SIMD technique illustrated in Example 6-46 consists of the following phases:

- Dword-to-byte packing and byte-array sequencing: The stream of dword elements is reduced to byte streams with each iteration handling 32 elements. The two resulting 16-byte vectors are sequenced to enable 4-way bit-stitching using PSLLD and PSRLD instructions.

1. “SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units”, T. Willhalm, et. al., Proceedings of the VLDB Endowment, Vol. 2, #1, August 2009.
2. "Super-Scalar RAM-CPU Cache Compression," M. Zukowski, et, al, Data Engineering, International Conference, vol. 0, no. 0, pp. 59, 2006.

Example 6-46. Compress 32-bit Integers into 5-bit Buckets

```
;
static __declspec(align(16))  short mask_dw_5b[16] =  // 5-bit mask for 4 way bit-packing via dword
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0};  // packed shift
static  __declspec(align(16))  short sprdb_0_5_10_15[8] =  // shuffle control to re-arrange
{ 0xff00, 0xffff, 0x04ff, 0xffff, 0xffff, 0xff08, 0xffff, 0x0cff}; // bytes 0, 4, 8, 12 to gap positions at 0, 5, 10, 15

void RDRpack32x4_sse(int *src, int cnt, char * out)

int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb;
__m128i msk4 ;
__m128i sprd4 = _mm_loadu_si128( (__m128i*) &sprdb_0_5_10_15[0]);
    switch( bucket_width) {
    case 5:j= 0;


     msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
    // process 32 elements in each iteration
     for (i = 0; i < cnt; i+= 32)  {
          b0 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i]), _mm_loadu_si128( (__m128i*)
&src[i+4]));
          b1 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+8]), _mm_loadu_si128( (__m128i*)
&src[i+12]));
          b2 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+16]), _mm_loadu_si128( (__m128i*)
&src[i+20]));
          b3 = _mm_packus_epi32(_mm_loadu_si128( (__m128i*) &src[i+24]), _mm_loadu_si128( (__m128i*)
&src[i+28]));
          c0 = _mm_packus_epi16( _mm_unpacklo_epi64(b0, b1), _mm_unpacklo_epi64(b2, b3));
          // c0 contains bytes: 0-3, 8-11, 16-19, 24-27  elements
          c1 = _mm_packus_epi16( _mm_unpackhi_epi64(b0, b1), _mm_unpackhi_epi64(b2, b3));
          // c1 contains bytes: 4-7, 12-15, 20-23, 28-31
     b0 = _mm_and_si128( c0, msk4);          // keep lowest 5 bits in each way/dword
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 5));
          b0 = _mm_or_si128( b0, b1);                    // add next 5 bits to each way/dword
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 10));
          b0 = _mm_or_si128( b0, b1);
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 9), _mm_slli_epi32(msk4, 15));
          b0 = _mm_or_si128( b0, b1);
          b0 = _mm_and_si128( c0, msk4);          // keep lowest 5 bits in each way/dword
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 5));
          b0 = _mm_or_si128( b0, b1);                    // add next 5 bits to each way/dword
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 10));
          b0 = _mm_or_si128( b0, b1);
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 9), _mm_slli_epi32(msk4, 15));
          b0 = _mm_or_si128( b0, b1);
          b1 = _mm_and_si128( _mm_slli_epi32(c1, 20), _mm_slli_epi32(msk4, 20));
          b0 = _mm_or_si128( b0, b1);
```

Example 6-46. Compress 32-bit Integers into 5-bit Buckets (Contd.)

```
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 20), _mm_slli_epi32(msk4, 20));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 17), _mm_slli_epi32(msk4, 25));
        b0 = _mm_or_si128( b0, b1);
        b1 = _mm_and_si128( _mm_slli_epi32(c1, 14), _mm_slli_epi32(msk4, 30));
        b0 = _mm_or_si128( b0, b1);    // add next 2 bits from each dword channel, xmm full
        *(int*)&out[j] = _mm_cvtsi128_si32( b0);// the first dword is compressed and ready
        // re-distribute the remaining 3 dword and add gap bytes to store remained bits
        b0 = _mm_shuffle_epi8(b0, gap4x3);
        b1 = _mm_and_si128( _mm_srli_epi32(c1, 18), _mm_srli_epi32(msk4, 2)); // do 4-way packing of the next 3 bits
        b2 = _mm_and_si128( _mm_srli_epi32(c1, 21), _mm_slli_epi32(msk4, 3));
        b1 = _mm_or_si128( b1, b2); //5th byte compressed at bytes 0, 4, 8, 12
        // shuffle the fifth byte result to byte offsets of 0, 5, 10, 15
        b0 = _mm_or_si128( b0, _mm_shuffle_epi8(b1, sprd4));
        _mm_storeu_si128( (__m128i *) &out[j+4] , b0);
        j += bucket_width*4;
    }
    // handle remainder if cnt is not multiples of 32
    break;
   }
}
```

- Four-way bit stitching: In each way (dword) of the destination, 5 bits are packed consecutively from the corresponding byte element that contains 5 non-zero bit patterns. Since each dword destination will be completely filled up by the contents of 7 consecutive elements, the remaining three bits of the 7th element and the 8th element are done separately in a similar 4-way stitching operation but require the assistance of shuffle operations.

Example 6-47 shows the reverse operation of decompressing consecutively packed 5-bit buckets into 32-bit data elements.

Example 6-47. Decompression of a Stream of 5-bit Integers into 32-bit Elements

```
;
static __declspec(align(16))  short mask_dw_5b[16] =  // 5-bit mask for 4 way bit-packing via dword
{0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0, 0x1f, 0x0};  // packed shift
static  __declspec(align(16))  short pack_dw_4x3[8] =  // pack 3 dwords 1-4, 6-9, 11-14
{ 0xffff, 0xffff, 0x0201, 0x0403, 0x0706, 0x0908, 0xc0b, 0x0e0d}; // to vacate bytes 0-3
static  __declspec(align(16))  short packb_0_5_10_15[8] =  // shuffle control to re-arrange bytes
{ 0xffff, 0x0ff, 0xffff, 0x5ff, 0xffff, 0xaff, 0xffff, 0x0fff}; // 0, 5, 10, 15 to gap positions at  3, 7, 11, 15

void RDRunpack32x4_sse(char *src, int cnt, int * out)
{int i, j;
__m128i a0, a1, a2, a3, c0, c1, b0, b1, b2, b3, bb, d0, d1, d2, d3;
__m128i msk4 ;
__m128i pck4 = _mm_loadu_si128( (__m128i*) &packb_0_5_10_15[0]);
__m128i pckdw3 = _mm_loadu_si128( (__m128i*) &pack_dw_4x3[0]);
```

Example 6-47. Decompression of a Stream of 5-bit Integers into 32-bit Elements (Contd.)

```
    switch( bucket_width) {
    case 5:j= 0;
     msk4 = _mm_loadu_si128( (__m128i*) &mask_dw_5b[0]);
for (i = 0; i < cnt; i+= 32)  {
          a1 =  _mm_loadu_si128( (__m128i*) &src[j +4]);
          // pick up bytes 4, 9, 14, 19 and shuffle into offset 3, 7, 11, 15
          c0 = _mm_shuffle_epi8(a1, pck4);
          b1 = _mm_and_si128( _mm_srli_epi32(c0, 3), _mm_slli_epi32(msk4, 24));
          // put 3 unaligned dword 1-4, 6-9, 11-14 to vacate bytes 0-3
          a1 = _mm_shuffle_epi8(a1, pckdw3);
          b0 = _mm_and_si128( _mm_srli_epi32(c0, 6), _mm_slli_epi32(msk4, 16));
          a0 = _mm_cvtsi32_si128(  *(int *)&src[j ]);
          b1 = _mm_or_si128( b0, b1);   // finished decompress source bytes 4, 9, 14, 19
          a0 = _mm_or_si128( a0, a1);  // bytes 0-16 contain compressed bits
          b0 = _mm_and_si128( _mm_srli_epi32(a0, 14), _mm_slli_epi32(msk4, 16));
          b1 = _mm_or_si128( b0, b1);
          b0 = _mm_and_si128( _mm_srli_epi32(a0, 17), _mm_slli_epi32(msk4, 8));
          b1 = _mm_or_si128( b0, b1);
          b0 = _mm_and_si128( _mm_srli_epi32(a0, 20), msk4);
          b1 = _mm_or_si128( b0, b1);// b1 now full with decompressed  4-7,12-15,20-23,28-31
          _mm_storeu_si128( (__m128i *) &out[i+4] , _mm_cvtepu8_epi32(b1));
          b0 = _mm_and_si128( _mm_slli_epi32(a0, 9), _mm_slli_epi32(msk4, 24));
          c0 = _mm_and_si128( _mm_slli_epi32(a0, 6), _mm_slli_epi32(msk4, 16));
          b0 = _mm_or_si128( b0, c0);
          _mm_storeu_si128( (__m128i *) &out[i+12] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 4)));
          c0 = _mm_and_si128( _mm_slli_epi32(a0, 3), _mm_slli_epi32(msk4, 8));
          _mm_storeu_si128( (__m128i *) &out[i+20] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 8)));
          b0 = _mm_or_si128( b0, c0);
          _mm_storeu_si128( (__m128i *) &out[i+28] , _mm_cvtepu8_epi32(_mm_srli_si128(b1, 12)));
          c0 = _mm_and_si128( a0, msk4);
          b0 = _mm_or_si128( b0, c0);// b0 now full with decompressed  0-3,8-11,16-19,24-27

          _mm_storeu_si128( (__m128i *) &out[i] , _mm_cvtepu8_epi32(b0));
          _mm_storeu_si128( (__m128i *) &out[i+8] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 4)));
          _mm_storeu_si128( (__m128i *) &out[i+16] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 8)));
          _mm_storeu_si128( (__m128i *) &out[i+24] , _mm_cvtepu8_epi32(_mm_srli_si128(b0, 12)));
         j += g_bwidth*4;
     }
     break;
    }
}
```

Compression/decompression of integers for dynamic range that are non-power-of-2 can generally use similar mask/packed shift/stitch technique with additional adaptation of the horizontal rearrangement of partially stitched vectors. The increase in throughput relative to using general-purpose scalar instructions will depend on implementation and bucket width.

When compiled with the "/O2" option on an Intel Compiler, the compression throughput can reach 6 Bytes/cycle on Sandy Bridge microarchitecture, and the throughput varies little for working set sizes due to the streaming data

access pattern and the effectiveness of hardware prefetchers. The decompression throughput of the above example is more than 5 Bytes/cycle at full utilization, allowing a database query engine to partition CPU utilization effectively to allocate a small fraction for on-the-fly decompression to feed vectorized query computation.

The decompression throughput increase using a SIMD light-weight compression technique offers database architects new degrees of freedom to relocate critical performance bottlenecks from a lower-throughput technology (disk I/O, DRAM) to a faster pipeline.

CHAPTER 7
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

This chapter discusses rules for optimizing the single-instruction, multiple-data (SIMD) floating-point instructions available in Intel® SSE, Intel® SSE2, Intel® SSE3, and Intel® SSE4.1. The chapter also provides examples illustrating the optimization techniques for single-precision and double-precision SIMD floating-point applications.

7.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE

The rules and suggestions in this section help optimize floating-point code containing SIMD floating-point instructions. Generally, it is essential to understand and balance port utilization to create efficient SIMD floating-point code. Basic rules and suggestions include the following:

- Follow all guidelines in Chapter 3, "General Optimization Guidelines".
- Mask exceptions to achieve higher performance. When exceptions are unmasked, software performance is slower.
- Utilize the flush-to-zero and denormals-are-zero modes for higher performance to avoid the penalty of dealing with denormals and underflows.
- Use the reciprocal instructions followed by iteration for increased accuracy. These instructions yield reduced accuracy but execute much faster. Note the following:
  - If reduced accuracy is acceptable, use them with no iteration.
  - If near full accuracy is needed, use a Newton-Raphson iteration.
  - If full accuracy is needed, then use divide and square root, which provide more accuracy, but slow down performance.

7.2 PLANNING CONSIDERATIONS

Whether adapting an existing application or creating a new one, using SIMD floating-point instructions to achieve optimum performance gain requires programmers to consider several issues. When choosing candidates for optimization, look for code segments that are computationally intensive and floating-point intensive. Also, consider efficient use of the cache architecture.

The sections that follow answer the questions that should be raised before implementation:

- Can data layout be arranged to increase parallelism or cache utilization?
- Which part of the code benefits from SIMD floating-point instructions?
- Is the current algorithm the most appropriate for SIMD floating-point instructions?
- Is the code floating-point intensive?
- Do single-precision floating-point or double-precision floating-point computations provide enough range and precision?
- Does the result of computation affected by enabling flush-to-zero or denormals-to-zero modes?
- Is the data arranged for efficient utilization of the SIMD floating-point registers?
- Is this application targeted for processors without SIMD floating-point instructions?

See Section 5.2.

7.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT

Because the XMM registers used for SIMD floating-point computations are separate registers and are not mapped to the existing x87 floating-point stack, SIMD floating-point code can be mixed with x87 floating-point or 64-bit SIMD integer code.

With Intel Core microarchitecture, 128-bit SIMD integer instructions provide substantially higher efficiency than 64-bit SIMD integer instructions. Software should favor using SIMD floating-point and integer SIMD instructions with XMM registers where possible.

7.4 SCALAR FLOATING-POINT CODE

SIMD floating-point instructions operate only on the lowest order element in the SIMD register. These instructions are known as scalar instructions. They allow the XMM registers to be used for general-purpose floating-point computations.

In terms of performance, scalar floating-point code can be equivalent to or exceed x87 floating-point code and has the following advantages:

- SIMD floating-point code uses a flat register model, whereas x87 floating-point code uses a stack model. Using scalar floating-point code eliminates the need to use FXCH instructions.
- Mixing with MMX technology code without penalty.
- Flush-to-zero mode.
- Shorter latencies than x87 floating-point.

When using scalar floating-point instructions, it is unnecessary to ensure that the data appears in vector form. However, the optimizations for alignment, scheduling, instruction selection, and other optimizations covered in Chapter 3, “General Optimization Guidelines” and Chapter 5, “Coding for SIMD Architectures” should be observed.

7.5 DATA ALIGNMENT

SIMD floating-point data is 16-byte aligned. Referencing unaligned 128-bit SIMD floating-point data will result in an exception unless MOVUPS or MOVUPD (move unaligned packed single or unaligned packed double) is used. The unaligned instructions used on aligned or unaligned data will also suffer a performance penalty relative to aligned accesses.

See also: Section 5.4.

7.5.1 DATA ARRANGEMENT

Because SSE and SSE2 incorporate SIMD architecture, arranging data to use the SIMD registers fully produces optimum performance. This implies contiguous data for processing, which leads to fewer cache misses. Correct data arrangement can quadruple data throughput using SSE, or double throughput when using SSE2. Performance gains can occur because four data elements can be loaded with 128-bit load instructions into XMM registers using SSE (MOVAPS). Similarly, two data elements can be loaded with 128-bit load instructions into XMM registers using SSE2 (MOVAPD).

Refer to Section 5.4 for data arrangement recommendations. Duplicating and padding techniques overcome misalignment problems that in some data structures and arrangements. This increases the data space but avoids penalties for misaligned data access.

For some applications (3D geometry, for example), traditional data arrangement requires some changes to use the SIMD registers and parallel techniques fully. Traditionally, the data layout has been an array of structures (AoS). A new data layout has been proposed to fully use the SIMD registers in such applications: a structure of arrays (SoA) resulting in more optimized performance.

7.5.1.1 Vertical versus Horizontal Computation

Most floating-point arithmetic instructions in SSE/SSE2 provide a more significant performance gain on vertical data processing for parallel data elements. This means that each element of the destination results from an arithmetic operation performed from the source elements in the same vertical position (Figure 7-1).

To supplement these homogeneous arithmetic operations on parallel data elements, SSE and SSE2 provide data movement instructions (e.g., SHUFPS, UNPCKLPS, UNPCKHPS, MOVLHPS, MOVHLPS, etc.) that facilitate moving data elements horizontally.

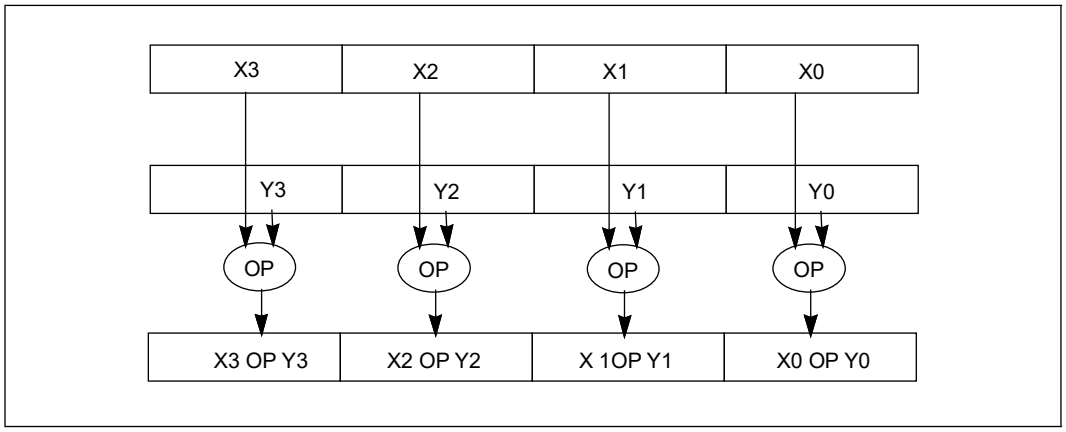


Figure 7-1. Homogeneous Operation on Parallel Data Elements

The organization of structured data significantly impacts SIMD programming efficiency and performance. This can be illustrated using two common type of data structure organizations:

- Array of Structure (AoS): This refers to arranging an array of data structures. Within the data structure, each member is a scalar. This is shown in Figure 7-2. Typically, a repetitive computation sequence is applied to each element of an array, i.e., a data structure. The computational sequence for the scalar members of the structure is likely to be non-homogeneous within each iteration. AoS is generally associated with a horizontal computation model.

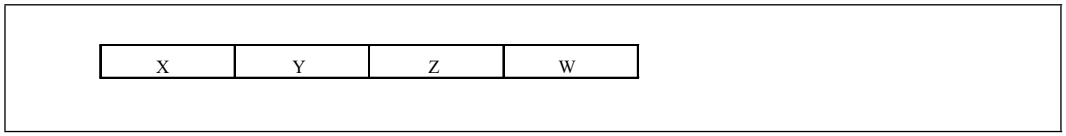


Figure 7-2. Horizontal Computation Model

- Structure of Array (SoA): Here, each member of the data structure is an array. Each element of the array is a scalar. This is shown in Table 7-1. The repetitive computational sequence is applied to scalar elements and homogeneous operation can be easily achieved across consecutive iterations within the same structural member. Consequently, SoA is generally amenable to the vertical computation model.

Table 7-1. SoA Form of Representing Vertices Data

Vx array	X1	X2	X3	X4	.....	Xn
Vy array	Y1	Y2	Y3	Y4	.....	Yn
Vz array	Z1	Z2	Z3	Y4	.....	Zn
Vw array	W1	W2	W3	W4	.....	Wn

SIMD instructions with vertical computation on the SoA arrangement can achieve higher efficiency and performance than AoS and horizontal computation. This can be seen with dot-product operation on vectors. The dot product operation on the SoA arrangement is shown in Figure 7-3.

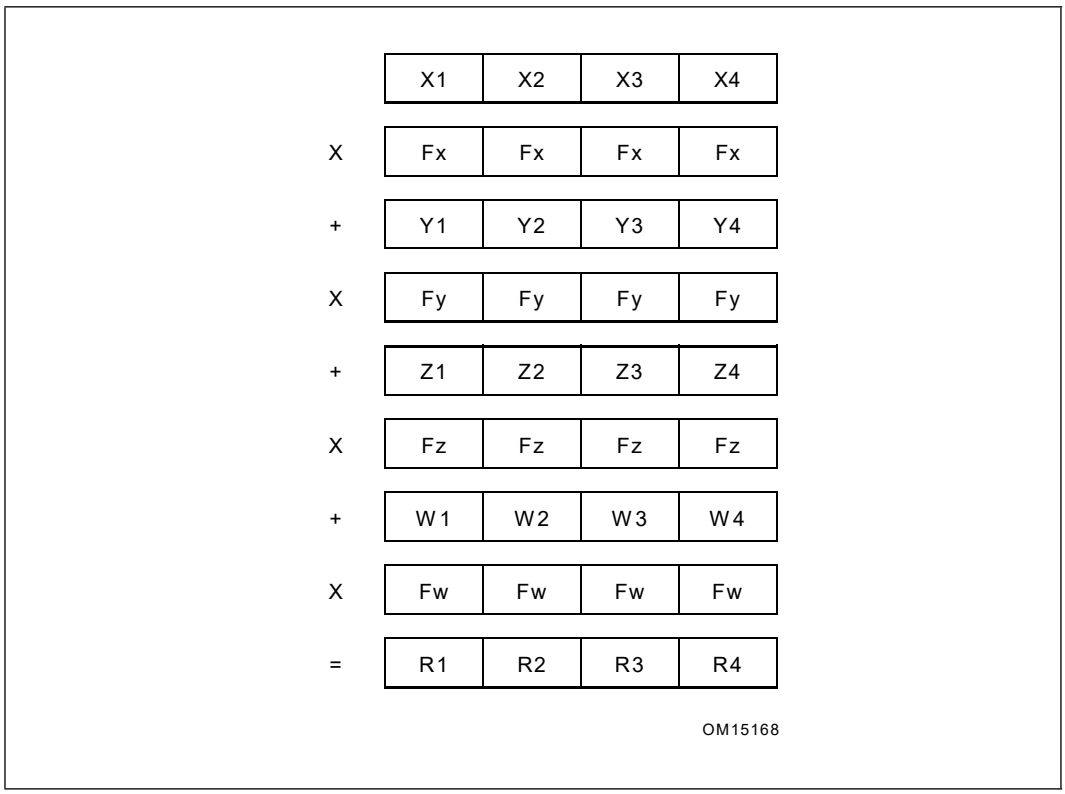


Figure 7-3. Dot Product Operation

Example 7-1 shows how one result would be computed for seven instructions if the data were organized as AoS and using SSE alone: four results would require 28 instructions.

Example 7-1. Pseudocode for Horizontal (xyz, AoS) Computation

```
mulps     ; x*x', y*y', z*z'
movaps    ; reg->reg move, since next steps overwrite
shufps    ; get b,a,d,c from a,b,c,d
addps     ; get a+b,a+b,c+d,c+d
movaps    ; reg->reg move
shufps    ; get c+d,c+d,a+b,a+b from prior addps
addps     ; get a+b+c+d,a+b+c+d,a+b+c+d,a+b+c+d
```

Now consider the case when the data is organized as SoA. Example 7-2 demonstrates how four results are computed for five instructions.

Example 7-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation

```
mulps   ; x*x' for all 4 x-components of 4 vertices
mulps   ; y*y' for all 4 y-components of 4 vertices
mulps   ; z*z' for all 4 z-components of 4 vertices
addps   ; x*x' + y*y'
addps   ; x*x'+y*y'+z*z'
```

For the most efficient use of the four component-wide registers, reorganizing the data into the SoA format yields increased throughput and hence much better performance for the instructions used.

This simple example shows that vertical computation can yield 100% use of the available SIMD registers to produce four results. Note that results may vary for other situations. Suppose the data structures are represented in a format that is not “friendly” to vertical computation. In that case, it can be rearranged “on the fly” to facilitate better utilization of the SIMD registers. This operation is referred to as a “swizzling” operation. The reverse operation is referred to as “deswizzling.”

7.5.1.2 Data Swizzling

Swizzling data from SoA to AoS format can apply to multiple application domains, including 3D geometry, video and imaging. Two different swizzling techniques can be adapted to handle floating-point and integer data. Example 7-3 illustrates a swizzle function that uses SHUFPS, MOVLHPS, and MOVHLPS instructions.

Example 7-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS

```
typedef struct _VERTEX_AOS {
    float x, y, z, color;
} Vertex_aos;                          // AoS structure declaration
typedef struct _VERTEX_SOA {
    float x[4], float y[4], float z[4];
    float color[4];
} Vertex_soa;                          // SoA structure declaration
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
```

Example 7-3. Swizzling Data (Contd.)Using SHUFPS, MOVLHPS, MOVHLPS (Contd.)

```
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov  rbx, in                    // get structure addresses
        mov  rdx, out
        movaps  xmm1, [rbx ]            // w0 z0 y0 x0
        movaps  xmm2, [rbx + 16]        // w1 z1 y1 x1
        movaps  xmm3, [rbx + 32]        // w2 z2 y2 x2
        movaps  xmm4, [rbx + 48]        // w3 z3 y2 x3
        movaps  xmm7, xmm4              // xmm7= w3 z3 y3 x3
        movhlps  xmm7, xmm3             // xmm7= w3 z3 w2 z2
        movaps  xmm6, xmm2              // xmm6= w1 z1 y1 x1
        movlhps  xmm3, xmm4             // xmm3= y3 x3 y1 x1
        movhlps  xmm2, xmm1             // xmm2= w1 z1 w0 z0
        movlhps  xmm1, xmm6             // xmm1= y1 x1 y0 x0
        movaps  xmm6, xmm2                  // xmm6= w1 z1 w0 z0
        movaps  xmm5, xmm1                  // xmm5= y1 x1 y0 x0
        shufps   xmm2, xmm7, 0xDDh          // xmm2= w3 w2 w1 w0 => W
        shufps   xmm1, xmm3, 0x88h          // xmm1= x3 x2 x1 x0 => X
        shufps   xmm5, xmm3, 0xDDh          // xmm5= y3 y2 y1 y0 => Y
        shufps   xmm6, xmm7, 0x88h          // xmm6= z3 z2 z1 z0 => Z
        movaps  [rdx], xmm1                 // store X
        movaps  [rdx+16], xmm5              // store Y
        movaps  [rdx+32], xmm6              // store Z
        movaps  [rdx+48], xmm2              // store W
    }
}
```

Example 7-4 shows a similar data-swizzling algorithm using SIMD instructions in the integer domain.

Example 7-4. Swizzling Data Using UNPCKxxx Instructions

```
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
// in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
// SWIZZLE XYZW --> XXXX
asm {
        mov rbx, in                     // get structure addresses
        mov rdx, out
        movdqa      xmm1, [rbx + 0*16]    //w0 z0 y0 x0
        movdqa      xmm2, [rbx + 1*16]    //w1 z1 y1 x1
        movdqa      xmm3, [rbx + 2*16]    //w2 z2 y2 x2
        movdqa      xmm4, [rbx + 3*16]    //w3 z3 y3 x3
        movdqa      xmm5, xmm1
        punpckldq   xmm1, xmm2          // y1 y0 x1 x0
        punpckhdq   xmm5, xmm2          // w1 w0 z1 z0
        movdqa      xmm2, xmm3
```

Example 7-4. Swizzling Data Using UNPCKxxx Instructions (Contd.)

```
        punpckldq   xmm3, xmm4          // y3 y2 x3 x2
        punpckhdq   xmm2, xmm4          // w3 w2 z3 z2
        movdqa      xmm4, xmm1
        punpcklqdq  xmm1, xmm3          // x3 x2 x1 x0
        punpckhqdq  xmm4, xmm3          // y3 y2 y1 y0
        movdqa      xmm3, xmm5
        punpcklqdq  xmm5, xmm2          // z3 z2 z1 z0
        punpckhqdq  xmm3, xmm2          // w3 w2 w1 w0
        movdqa      [rdx+0*16], xmm1    //x3 x2 x1 x0
        movdqa      [rdx+1*16], xmm4    //y3 y2 y1 y0
        movdqa      [rdx+2*16], xmm5    //z3 z2 z1 z0
        movdqa      [rdx+3*16], xmm3    //w3 w2 w1 w0
}
```

The technique in Example 7-3 (loading 16 bytes, using SHUFPS and copying halves of XMM registers) is preferable over an alternate approach of loading halves of each vector using MOVLPS/MOVHPS on newer microarchitectures. This is because loading 8 bytes using MOVLPS/MOVHPS can create code dependency and reduce the throughput of the execution engine.

The performance considerations of Example 7-3, and Example 7-4 often depend on each microarchitecture's characteristics. For example, in Intel Core microarchitecture, executing a SHUFPS tend to be slower than a PUNPCKxxx instruction. In Enhanced Intel Core microarchitecture, SHUFPS and PUNPCKxxx instruction execute with one cycle throughput due to the 128-bit shuffle execution unit. The next important consideration is that only one port can execute PUNPCKxxx rather than MOVLHPS/MOVHLPS executing on multiple ports. The performance of both techniques improves on Intel Core microarchitecture over previous microarchitectures due to 3 ports for executing SIMD instructions. Both techniques further improve the Enhanced Intel Core microarchitecture due to the 128-bit shuffle unit.

7.5.1.3 Data Deswizzling

In the deswizzle operation, we want to arrange the SoA format back into AoS format so the XXXX, YYYY, and ZZZZ are rearranged and stored in memory as XYZ. Example 7-5 illustrates one deswizzle function for floating-point data.

Example 7-5. Deswizzling Single-Precision SIMD Data

```
void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
  __asm {
    mov         rcx, in             // load structure addresses
    mov         rdx, out
    movaps      xmm0, [rcx]         //x3 x2 x1 x0
    movaps      xmm1, [rcx + 16]    //y3 y2 y1 y0
    movaps      xmm2, [rcx + 32]    //z3 z2 z1 z0
    movaps      xmm3, [rcx + 48]    //w3 w2 w1 w0
    movaps      xmm5, xmm0
    movaps      xmm7, xmm2
    unpcklps    xmm0, xmm1          // y1 x1 y0 x0
    unpcklps    xmm2, xmm3          // w1 z1 w0 z0
```

Example 7-5. Deswizzling Single-Precision SIMD Data (Contd.)

```
    movdqa      xmm4, xmm0
    movlhps     xmm0, xmm2          // w0 z0 y0 x0
    movhlps     xmm2, xmm4          // w1 z1 y1 x1
    unpckhps    xmm5, xmm1          // y3 x3 y2 x2
    unpckhps    xmm7, xmm3          // w3 z3 w2 z2
    movdqa      xmm6, xmm5
    movlhps     xmm5, xmm7          // w2 z2 y2 x2
    movhlps     xmm7, xmm6          // w3 z3 y3 x3
    movaps      [rdx+0*16], xmm0    //w0 z0 y0 x0
    movaps      [rdx+1*16], xmm2    //w1 z1 y1 x1
    movaps      [rdx+2*16], xmm5    //w2 z2 y2 x2
    movaps      [rdx+3*16], xmm7    //w3 z3 y3 x3
  }
}
```

Example 7-6 shows a similar deswizzle function using SIMD integer instructions. Both techniques demonstrate loading 16 bytes and performing horizontal data movement in registers. This approach is likely more efficient than alternative techniques of storing 8-byte halves of XMM registers using MOVLPS and MOVHPS.

Example 7-6. Deswizzling Data Using SIMD Integer Instructions

```
void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
        //---deswizzling---rgb---
        // assume: xmm0=rrrr, xmm1=gggg, xmm2=bbbb, xmm3=aaaa
    mov         rcx, in                         // load structure addresses
    mov         rdx, out

    movdqa      xmm0, [rcx]                     // load r4 r3 r2 r1 => xmm0
    movdqa      xmm1, [rcx+16]                  // load g4 g3 g2 g1 => xmm1
    movdqa      xmm2, [rcx+32]                  // load b4 b3 b2 b1 => xmm2
    movdqa      xmm3, [rcx+48]                  // load a4 a3 a2 a1 => xmm3

// Start deswizzling here
    movdqa      xmm5, xmm0
    movdqa      xmm7, xmm2
    punpckldq   xmm0, xmm1                      //g2 r2 g1 r1
    punpckldq   xmm2, xmm3                      //a2 b2 a1 b1
    movdqa      xmm4, xmm0
    punpcklqdq  xmm0, xmm2                      // a1 b1 g1 r1 => v1
    punpckhqdq  xmm4, xmm2                      // a2 b2 g2 r2 => v2
    punpckhdq   xmm5, xmm1                      // g4 r4 g3 r3
    punpckhdq   xmm7, xmm3                      // a4 b4 a3 b3
    movdqa      xmm6, xmm5
    punpcklqdq  xmm5, xmm7                      // a3 b3 g3 r3 => v3
    punpckhqdq  xmm6, xmm7                      // a4 b4 g4 r4 => v4
    movdqa      [rdx], xmm0                     // v1
    movdqa      [rdx+16], xmm4                  // v2
```

Example 7-6. Deswizzling Data Using SIMD Integer (Contd.)Instructions

```
    movdqa      [rdx+32], xmm5              // v3
    movdqa      [rdx+48], xmm6              // v4

// DESWIZZLING ENDS HERE
    }
}
```

7.5.1.4 Horizontal ADD Using SSE

Although vertical computations generally use SIMD performance better than horizontal computations, code must use a horizontal operation in some cases.

MOVLHPS/MOVHLPS and shuffle can be used to sum data horizontally. For example, starting with four 128-bit registers, to sum up each register horizontally while having the final results in one register, use the MOVLHPS/MOVHLPS to align the upper and lower parts of each register. This allows you to use a vertical add. With the resulting partial horizontal summation, full summation follows easily.

Figure 7-4 presents a horizontal add using MOVHLPS/MOVLHPS. Example 7-7 and Example 7-8 provide the code for this operation.

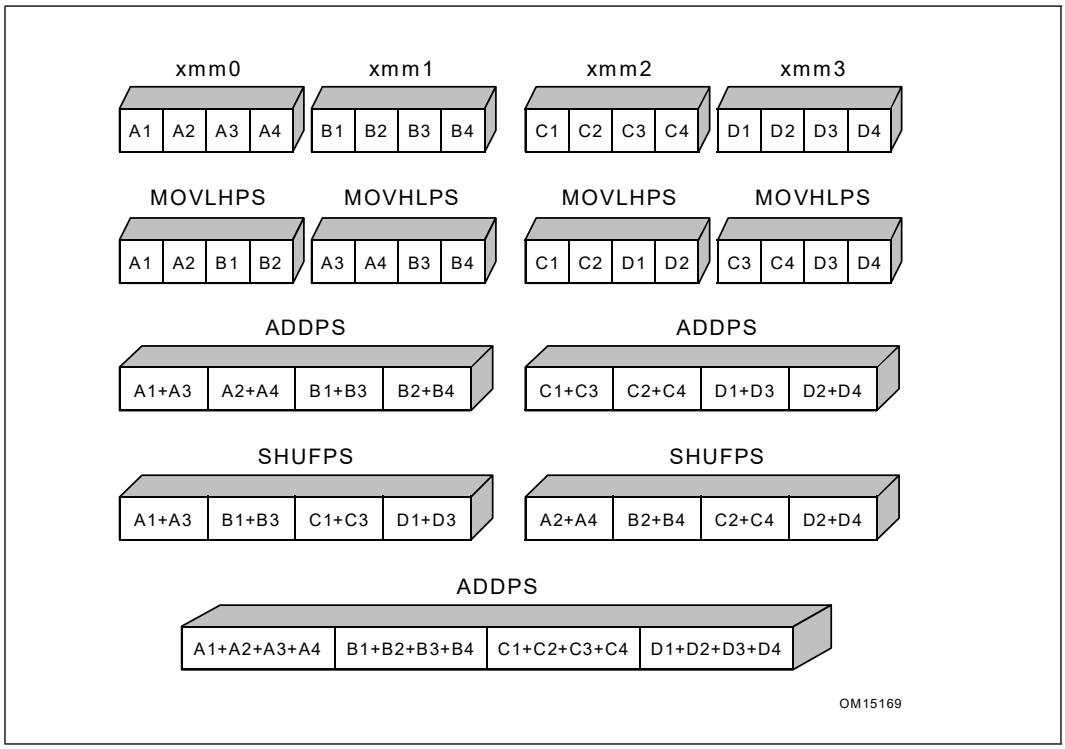


Figure 7-4. Horizontal Add Using MOVHLPS/MOVLHPS

Example 7-7. Horizontal Add Using MOVHLPS/MOVLHPS

```
void horiz_add(Vertex_soa *in, float *out) {
 __asm {
    mov         rcx, in                 // load structure addresses
    mov         rdx, out
    movaps      xmm0, [rcx]             // load A1 A2 A3 A4 => xmm0
    movaps      xmm1, [rcx+16]          // load B1 B2 B3 B4 => xmm1
    movaps      xmm2, [rcx+32]          // load C1 C2 C3 C4 => xmm2
    movaps      xmm3, [rcx+48]          // load D1 D2 D3 D4 => xmm3
 // START HORIZONTAL ADD
    movaps      xmm5, xmm0              // xmm5= A1,A2,A3,A4
    movlhps     xmm5, xmm1              // xmm5= A1,A2,B1,B2
    movhlps     xmm1, xmm0              // xmm1= A3,A4,B3,B4
    addps       xmm5, xmm1              // xmm5= A1+A3,A2+A4,B1+B3,B2+B4
    movaps      xmm4, xmm2
    movlhps     xmm2, xmm3              // xmm2= C1,C2,D1,D2
    movhlps     xmm3, xmm4              // xmm3= C3,C4,D3,D4
    addps       xmm3, xmm2              // xmm3= C1+C3,C2+C4,D1+D3,D2+D4
    movaps      xmm6, xmm3              // xmm6= C1+C3,C2+C4,D1+D3,D2+D4
    shufps      xmm3, xmm5, 0xDD        //xmm6=A1+A3,B1+B3,C1+C3,D1+D3
    shufps      xmm5, xmm6, 0x88        // xmm5= A2+A4,B2+B4,C2+C4,D2+D4
    addps       xmm6, xmm5              // xmm6= D,C,B,A
 // END HORIZONTAL ADD
   movaps    [rdx], xmm6
 }
}
```

Example 7-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS

```
void horiz_add_intrin(Vertex_soa *in, float *out)
{
 __m128 v, v2, v3, v4;
 __m128 tmm0,tmm1,tmm2,tmm3,tmm4,tmm5,tmm6;
                                                  // Temporary variables
    tmm0 = _mm_load_ps(in->x);                    // tmm0 = A1 A2 A3 A4
    tmm1 = _mm_load_ps(in->y);                    // tmm1 = B1 B2 B3 B4
    tmm2 = _mm_load_ps(in->z);                    // tmm2 = C1 C2 C3 C4
    tmm3 = _mm_load_ps(in->w);                    // tmm3 = D1 D2 D3 D4
    tmm5 = tmm0;                                  // tmm0 = A1 A2 A3 A4
    tmm5 = _mm_movelh_ps(tmm5, tmm1);             // tmm5 = A1 A2 B1 B2
    tmm1 = _mm_movehl_ps(tmm1, tmm0);             // tmm1 = A3 A4 B3 B4
    tmm5 = _mm_add_ps(tmm5, tmm1);                // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
    tmm4 = tmm2;
```

Example 7-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS (Contd.)

```
    tmm2 = _mm_movelh_ps(tmm2, tmm3);          // tmm2 = C1 C2 D1 D2
    tmm3 = _mm_movehl_ps(tmm3, tmm4);          // tmm3 = C3 C4 D3 D4
    tmm3 = _mm_add_ps(tmm3, tmm2);             // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = tmm3;                               // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);   // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
    tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);   // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
    tmm6 = _mm_add_ps(tmm6, tmm5);             // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
                                               // C1+C2+C3+C4 D1+D2+D3+D4
  _mm_store_ps(out, tmm6);
}
```

7.5.2 USE OF CVTTPS2PI/CVTTSS2SI INSTRUCTIONS

The CVTTPS2PI and CVTTSS2SI instructions implicitly encode the truncate/chop rounding mode in the instruction. They take precedence over the rounding mode specified in the MXCSR register. This behavior can eliminate the need to change the rounding mode from round-nearest, to truncate/chop, then return to round-nearest to resume computation.

Avoid frequent changes to the MXCSR register since a penalty associated with writing this register. Typically, when using CVTTPS2P/CVTTSS2SI, rounding control in MXCSR can always be set to round-nearest.

7.5.3 FLUSH-TO-ZERO AND DENORMALS-ARE-ZERO MODES

The flush-to-zero (FTZ) and denormals-are-zero (DAZ) modes are incompatible with IEEE Standard 754[1]. They are provided to improve performance for applications where underflow is common and generating a denormalized result is unnecessary.

See Section 3.9.2.

7.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES

Intel Core processors have a different microarchitecture than the Intel NetBurst microarchitecture. Intel Core microarchitecture offers significantly more efficient SIMD floating-point capability than previous microarchitectures. In addition, instruction latency and throughput of SSE3 instructions are improved considerably in Intel Core microarchitectures over previous microarchitectures.

7.6.1 DOT PRODUCT AND HORIZONTAL SIMD INSTRUCTIONS

Sometimes the AOS-type of data organization is more natural in many algebraic formulas. One typical example is the *dot product* operation. The dot product operation can be implemented using SSE/SSE2 instruction sets. SSE3 added a few horizontal add/subtract instructions for applications that rely on the horizontal computation model. SSE4.1

1. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229. https://ieeexplore.ieee.org/document/8766229

provides additional enhancement with instructions capable of directly evaluating dot product operations of vectors of 2, 3 or 4 components.

Example 7-9. Dot Product of Vector Length 4 Using SSE/SSE2

Using SSE/SSE2 to compute one dot product

```
movaps    xmm0, [rax]       // a4, a3, a2, a1
mulps     xmm0, [rax+16]    // a4*b4, a3*b3, a2*b2, a1*b1
movhlps   xmm1, xmm0        // X, X,  a4*b4, a3*b3, upper half not needed
addps     xmm0, xmm1        // X, X, a2*b2+a4*b4, a1*b1+a3*b3,
pshufd    xmm1, xmm0, 1     // X, X, X, a2*b2+a4*b4
addss     xmm0, xmm1        // a1*b1+a3*b3+a2*b2+a4*b4
movss     [rcx], xmm0
```

Example 7-10. Dot Product of Vector Length 4 Using SSE3

Using SSE3 to compute one dot product

```
movaps    xmm0, [rax]
mulps     xmm0, [rax+16]    // a4*b4, a3*b3, a2*b2, a1*b1
haddps    xmm0, xmm0        // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
movaps    xmm1, xmm0        // a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
psrlq     xmm0, 32          // 0, a4*b4+a3*b3, 0, a4*b4+a3*b3
addss     xmm0, xmm1        // -, -, -, a1*b1+a3*b3+a2*b2+a4*b4
movss     [rax], xmm0
```

Example 7-11. Dot Product of Vector Length 4 Using SSE4.1

Using SSE4.1 to compute one dot product

```
movaps    xmm0, [rax]
dpps      xmm0, [rax+16], 0xf1  // 0, 0, 0, a1*b1+a3*b3+a2*b2+a4*b4
movss     [rax], xmm0
```

Example 7-9, Example 7-10, and Example 7-11 compare the basic code sequence to compute one dot-product result for a pair of vectors.

The selection of an optimal sequence in conjunction with an application's memory access patterns may favor different approaches. For example, if each dot product result is immediately consumed by additional computational sequences, it may be more optimal to compare the relative speed of these different approaches. If dot products can be computed for an array of vectors and kept in the cache for subsequent computations, then more optimal choice may depend on the relative throughput of the sequence of instructions.

In Intel Core microarchitecture, Example 7-10 has higher throughput than Example 7-9. Due to the relatively longer latency of HADDPS, the speed of Example 7-10 is slightly slower than Example 7-9.

In Enhanced Intel Core microarchitecture, Example 7-11 is faster in both speed and throughput than Example 7-9 and Example 7-10. Although the latency of DPPS is also relatively long, it is compensated by the reduction of number of instructions in Example 7-11 to do the same amount of work.

Unrolling can further improve the throughput of each of three dot product implementations. Example 7-12 shows two unrolled versions using the basic SSE2 and SSE3 sequences. The SSE4.1 version can also be unrolled and using INSERTPS to pack four dot-product results.

Example 7-12. Unrolled Implementation of Four Dot Products

SSE2 Implementation			SSE3 Implementation	
movaps	xmm0, [rax]		movaps	xmm0, [rax]
mulps	xmm0, [rax+16]	;w0*w1 z0*z1 y0*y1 x0*x1	mulps	xmm0, [rax+16]
movaps	xmm2, [rax+32]		movaps	xmm1, [rax+32]
mulps	xmm2, [rax+16+32]	;w2*w3 z2*z3 y2*y3 x2*x3	mulps	xmm1, [rax+16+32]
movaps	xmm3, [rax+64]		movaps	xmm2, [rax+64]
mulps	xmm3, [rax+16+64]	;w4*w5 z4*z5 y4*y5 x4*x5	mulps	xmm2, [rax+16+64]
movaps	xmm4, [rax+96]		movaps	xmm3, [rax+96]
mulps	xmm4, [rax+16+96]	;w6*w7 z6*z7 y6*y7 x6*x7	mulps	xmm3, [rax+16+96]
movaps	xmm1, xmm0		haddps	xmm0, xmm1
unpcklps	xmm0, xmm2	; y2*y3 y0*y1 x2*x3 x0*x1	haddps	xmm2, xmm3
unpckhps	xmm1, xmm2	; w2*w3 w0*w1 z2*z3 z0*z1	haddps	xmm0, xmm2
movaps	xmm5, xmm3		movaps	[rcx], xmm0
unpcklps	xmm3, xmm4	; y6*y7 y4*y5 x6*x7 x4*x5		
unpckhps	xmm5, xmm4	; w6*w7 w4*w5 z6*z7 z4*z5		
addps	xmm0, xmm1			
addps	xmm5, xmm3			
movaps	xmm1, xmm5			
movhlps	xmm1, xmm0			
movlhps	xmm0, xmm5			
addps	xmm0, xmm1			
movaps	[rcx], xmm0			

7.6.2 VECTOR NORMALIZATION

Normalizing vectors is a common operation in many floating-point applications. Example 7-13 shows an example in C of normalizing an array of (x, y, z) vectors.

Example 7-13. Normalization of an Array of Vectors

```
for (i=0;i<CNT;i++)
{ float size = nodes[i].vec.dot();
    if (size != 0.0)
    { size = 1.0f/sqrtf(size); }
    else
    { size = 0.0; }
    nodes[i].vec.x *= size;
    nodes[i].vec.y *= size;
    nodes[i].vec.z *= size;
}
```

Example 7-14 shows an assembly sequence that normalizes the x, y, z components of a vector.

Example 7-14. Normalize (x, y, z) Components of an Array of Vectors Using SSE2

```
Vec3 *p = &nodes[i].vec;
__asm
{   mov     rax, p
    xorps   xmm2, xmm2
    movups  xmm1, [rax]         // loads the (x, y, z) of input vector plus x of next vector
    movaps  xmm7, xmm1          // save a copy of data from memory (to restore the unnormalized value)
    movaps  xmm5, _mask         // mask to select (x, y, z) values from an xmm register to normalize
    andps   xmm1, xmm5          // mask 1st 3 elements
    movaps  xmm6, xmm1          // save a copy of (x, y, z) to compute normalized vector later
    mulps   xmm1,xmm1           // 0, z*z, y*y, x*x
    pshufd  xmm3, xmm1, 0x1b    // x*x, y*y, z*z, 0
    addps   xmm1, xmm3          // x*x, z*z+y*y, z*z+y*y, x*x
    pshufd  xmm3, xmm1, 0x41    // z*z+y*y, x*x, x*x, z*z+y*y
    addps   xmm1, xmm3          // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd  xmm1, xmm2          // compare size to 0
    jz zero
    movaps  xmm3, xmm4          // preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
    sqrtps  xmm1, xmm1
    divps   xmm3, xmm1
    jmp     store
zero:
    movaps  xmm3, xmm2
store:
    mulps   xmm3, xmm6          //normalize the vector in the lower 3 elements
    andnps  xmm5, xmm7          // mask off the lower 3 elements to keep the un-normalized value
    orps    xmm3, xmm5          // order the un-normalized component after the normalized vector
    movaps  [rax], xmm3         // writes normalized x, y, z; followed by unmodified value
```

Example 7-15 shows an assembly sequence using SSE4.1 to normalizes the x, y, z components of a vector.

Example 7-15. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1

```
Vec3 *p = &nodes[i].vec;
__asm
{   mov     rax, p
    xorps   xmm2, xmm2
    movups  xmm1, [rax]          // loads the (x, y, z) of input vector plus x of next vector
    movaps  xmm7, xmm1           // save a copy of data from memory
    dpps    xmm1, xmm1, 0x7f     // x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z, x*x+y*y+z*z
    comisd  xmm1, xmm2           // compare size to 0
    jz zero
    movaps  xmm3, xmm4           // preloaded unitary vector (1.0, 1.0, 1.0, 1.0)
    sqrtps  xmm1, xmm1
    divps   xmm3, xmm1
    jmp     store
zero:
    movaps  xmm3, xmm2

store:
    mulps   xmm3, xmm6           //normalize the vector in the lower 3 elements
    blendps xmm3, xmm7, 0x8      // copy the un-normalized component next to the normalized vector
    movaps  [rax], xmm3
```

In Example 7-14 and Example 7-15, the throughput of these instruction sequences are basically limited by the long-latency instructions of DIVPS and SQRTPS. In Example 7-15, the use of DPPS replaces eight SSE2 instructions to evaluate and broadcast the dot-product result to four elements of an XMM register. This could result in improvement of the relative speed of Example 7-15 over Example 7-14.

7.6.3 USING HORIZONTAL SIMD INSTRUCTION SETS AND DATA LAYOUT

SSE and SSE2 provide packed add/subtract, multiply/divide instructions that are ideal for situations that can take advantage of vertical computation model, such as SOA data layout. SSE3 and SSE4.1 added horizontal SIMD instructions including horizontal add/subtract, dot-product operations. These more recent SIMD extensions provide tools to solve problems involving data layouts or operations that do not conform to the vertical SIMD computation model.

In this section, we consider a vector-matrix multiplication problem and discuss the relevant factors for choosing various horizontal SIMD instructions.

Example 7-16 shows the vector-matrix data layout in AOS, where the input and out vectors are stored as an array of structure.

Example 7-16. Data Organization in Memory for AOS Vector-Matrix Multiplication

```
Matrix M4x4 (pMat):              M00 M01 M02 M03
                                 M10 M11 M12 M13
                                 M20 M21 M22 M23
                                 M30 M31 M32 M33
4 input vertices V4x1 (pVert):   V0x V0y V0z V0w
                                 V1x V1y V1z V1w
                                 V2x V2y V2z V2w
                                 V3x V3y V3z V3w
Output vertices O4x1 (pOutVert): O0x O0y O0z O0w
                                 O1x O1y O1z O1w
                                 O2x O2y O2z O2w
                                 O3x O3y O3z O3w
```

Example 7-17 shows an example using HADDPS and MULPS to perform vector-matrix multiplication with data layout in AOS. After three HADDPS completing the summations of each output vector component, the output components are arranged in AOS.

Example 7-17. AOS Vector-Matrix Multiplication with HADDPS

```
    mov     rax, pMat
    mov     rbx, pVert
    mov     rcx, pOutVert
    xor                 rdx, rdx
    movaps  xmm5,[rax+16]       // load row M1?
    movaps  xmm6,[rax+2*16]     // load row M2?
    movaps  xmm7,[rax+3*16]     // load row M3?
lloop:
    movaps  xmm4, [rbx + rdx]   // load input vector
    movaps  xmm0, xmm4
    mulps   xmm0, [rax]         // m03*vw, m02*vz, m01*vy, m00*vx,
    movaps  xmm1, xmm4
    mulps   xmm1, xmm5          // m13*vw, m12*vz, m11*vy, m10*vx,

    movaps  xmm2, xmm4
    mulps   xmm2, xmm6          // m23*vw, m22*vz, m21*vy, m20*vx
    movaps  xmm3, xmm4
    mulps   xmm3, xmm7          // m33*vw, m32*vz, m31*vy, m30*vx,
    haddps  xmm0, xmm1
    haddps  xmm2, xmm3
    haddps  xmm0, xmm2
    movaps  [rcx + rdx], xmm0   // store a vector of length 4
    add     rdx, 16
    cmp     rdx, top
    jb lloop
```

Example 7-18 shows an example using DPPS to perform vector-matrix multiplication in AOS.

Example 7-18. AOS Vector-Matrix Multiplication with DPPS

```
mov     rax, pMat
    mov     rbx, pVert
    mov     rcx, pOutVert
    xor                     rdx, rdx
    movaps  xmm5,[rax+16]           // load row M1?
    movaps  xmm6,[rax+2*16]         // load row M2?
    movaps  xmm7,[rax+3*16]         // load row M3?
lloop:
    movaps  xmm4, [rbx + rdx]       // load input vector
    movaps  xmm0, xmm4
    dpps    xmm0, [rax], 0xf1       // calculate dot product of length 4, store to lowest dword
    movaps  xmm1, xmm4
    dpps    xmm1, xmm5, 0xf1
    movaps  xmm2, xmm4
    dpps    xmm2, xmm6, 0xf1
```

```
    movaps  xmm3, xmm4dpps xmm3, xmm7, 0xf1
    movss   [rcx + rdx + 0*4], xmm0   // store one element of vector length 4
    movss   [rcx + rdx + 1*4], xmm1
    movss   [rcx + rdx + 2*4], xmm2
    movss   [rcx + rdx + 3*4], xmm3
    add     rdx, 16
    cmp     rdx, top
    jb      lloop
```

Example 7-17 and Example 7-18 both work with AOS data layout using different horizontal processing techniques provided by SSE3 and SSE4.1. The effectiveness of either techniques will vary, depending on the degree of exposures of long-latency instruction in the inner loop, the overhead/efficiency of data movement, and the latency of HADDPS vs. DPPS.

On processors that support both HADDPS and DPPS, the choice between either technique may depend on application-specific considerations. If the output vectors are written back to memory directly in a batch situation, Example 7-17 may be preferable over Example 7-18, because the latency of DPPS is long and storing each output vector component individually is less than ideal for storing an array of vectors.

There may be partially-vectorizable situations that the individual output vector component is consumed immediately by other non-vectorizable computations. Then, using DPPS producing individual component may be more suitable than dispersing the packed output vector produced by three HADDPS as in Example 7-17.

7.6.3.1 SOA and Vector Matrix Multiplication

If the native data layout of a problem conforms to SOA, then vector-matrix multiply can be coded using MULPS, ADDPS without using the longer-latency horizontal arithmetic instructions, or packing scalar components into packed format (Example 7-18). To achieve higher throughput with SOA data layout, there are either prerequisite data preparation or swizzling/deswizzling on-the-fly that must be comprehended. For example, an SOA data layout for

vector-matrix multiplication is shown in Example 7-19. Each matrix element is replicated four times to minimize data movement overhead for producing packed results.

Example 7-19. Data Organization in Memory for SOA Vector-Matrix Multiplication

```
Matrix M16x4 (pMat):
        M00 M00 M00 M00 M01 M01 M01 M01 M02 M02 M02 M02 M03 M03 M03 M03
        M10 M10 M10 M10 M11 M11 M11 M11 M12 M12 M12 M12 M13 M13 M13 M13
        M20 M20 M20 M20 M21 M21 M21 M21 M22 M22 M22 M22 M23 M23 M23 M23
        M30 M30 M30 M30 M31 M31 M31 M31 M32 M32 M32 M32 M33 M33 M33 M33
4 input vertices V4x1 (pVert): V0x V1x V2x V3x
                               V0y V1y V2y V3y
                               V0z V1z V2z V3z
                               V0w V1w V2w V3w
Ouput vertices O4x1 (pOutVert): O0x O1x O2x O3x
                               O0y O1y O2y O3y
                               O0z O1z O2z O3z
                               O0w O1w O2w O3w
```

The corresponding vector-matrix multiply example in SOA (unrolled for four iteration of vectors) is shown in Example 7-20.

Example 7-20. Vector-Matrix Multiplication with Native SOA Data Layout

```
    mov     rbx, pVert
    mov     rcx, pOutVert
    xor     rdx, rdx
    movaps  xmm5,[rax + 16]     // load row M1?
    movaps  xmm6,[rax + 2*16]   // load row M2?
    movaps  xmm7,[rax + 3*16]   // load row M3?
lloop_vert:
    mov     rax, pMat
    xor     edi, edi
    movaps  xmm0, [rbx]         // load V3x, V2x, V1x, V0x
    movaps  xmm1, [rbx]         // load V3y, V2y, V1y, V0y
    movaps  xmm2, [rbx]         // load V3z, V2z, V1z, V0z
    movaps  xmm3, [rbx]         // load V3w, V2w, V1w, V0w
loop_mat:
    movaps  xmm4, [rax]         // m00, m00, m00, m00,
    mulps   xmm4, xmm0          // m00*V3x, m00*V2x, m00*V1x, m00*V0x,
    movaps  xmm4, [rax + 16]    // m01, m01, m01, m01,
    mulps   xmm5, xmm1          // m01*V3y, m01*V2y, m01*V1y, m01*V0y,
    addps   xmm4, xmm5
    movaps  xmm5, [rax + 32]    // m02, m02, m02, m02,
    mulps   xmm5, xmm2          // m02*V3z, m02*V2z, m02*V1z, m02*V0z,
    addps   xmm4, xmm5
    movaps  xmm5, [rax+ 48]     // m03, m03, m03, m03,
    mulps   xmm5, xmm3          // m03*V3w, m03*V2w, m03*V1w, m03*V0w,
    addps   xmm4, xmm5
    movaps  [rcx + rdx], xmm4
    add     rax, 64
    add     rdx, 16
    add     edi, 1
    cmp     edi, 4
    jb lloop_mat
    add     rbx, 64
    cmp     rdx, top
    jb lloop_vert
```

CHAPTER 8
INT8 DEEP LEARNING INFERENCE

This chapter describes INT8 as a data type for Deep learning Inference on Intel technology. The document covers both Intel® AVX-512 implementations and implementations using the new Intel® DL Boost Instructions.

The chapter is divided into several parts. The first part introduces INT8, and more specifically the Intel DL Boost instructions as the core data type and instructions for use in ML workloads. The second part discusses general methodologies and guidelines for efficient inference computation. The third part discusses optimizations specific to CNNs and the final part discusses optimizations specific to LSTM/RNNs.

When relevant, examples are provided with and without the new Intel DL Boost instruction set. In many cases (quantization, memory layout) there are steps that can be taken offline and steps that must be taken in runtime; we try to clearly state when each step is taken.

8.1 INTRODUCING INT8 AS DATA TYPE FOR DEEP LEARNING INFERENCE

Traditionally, deep learning is done with Single Precision Floating Point (F32) data type. Lately, INT8 has been used successfully for deep learning inference with a significant boost to performance and little loss of accuracy. The 4x narrower data type and the 3x more Intel® AVX-512 instructions required for INT8 MAC operation (vs. a single F32 FMA instruction) provide a net 1.33x nominal gain. Our experience with ResNet-50, Inception-Resnet v2, SRGAN and NMT on the Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture shows >1.5x speedup due to the INT8 smaller memory footprint. Section 8.2 describes Intel Deep Learning Boost instructions introduced in processors based on the Cascade Lake product, which further increase DL Inference performance.

We consider two use cases for DL Inference Workloads. The first usage is the Throughput Model, where elements (images, sentences) are processed regardless of how long it takes to process a single element. This usage model is usually appropriate for servers that process a bulk of images for classification or offline preparation of recommendations tailored to specific users. The second usage model is the Throughput at Latency Model where there is an upper limit for the time it is acceptable to compute a single element. This usage model is usually appropriate for online computation (language translation, real-time object detection, etc.).

8.2 INTRODUCING INTEL® DL BOOST

Intel® DL Boost instructions are a set of Intel AVX-512 instructions that are designed to speed up neural network workloads. These instructions are supported if CPUID.07H.0H:ECX.AVX512_VNNI[bit 11] = 1.

The following sections describe these new instructions and show a simple comparison to previous Intel AVX-512 code. Refer to the Intel® Architecture Instruction Set Extensions Programming Reference or the Intel® 64 and IA-32 Architectures Software Developer's Manual for complete instruction definitions (instructions with prefix VPDP).

8.2.1 MULTIPLY AND ADD UNSIGNED AND SIGNED BYTES (VPDPBUSD INSTRUCTION)

VPDPBUSD is an 8-bit integer multiply-accumulate vector operation into a vector of 32-bit integer accumulators. It accepts two source vector operands and one source/destination vector operand. The two source operands contain 8-bit integer data elements. The source/destination operand contains 32-bit data elements.

As an example, take 512-bit vector operands where each of the source operands contains 64 x 8-bit elements and the source/destination operand contains 16 x 32-bit elements.

The instruction splits the 64 elements in each source operand into 16 quadruples and multiplies vertically the four members of each quadruple, one from each source operand to create four 32-bit intermediate results. It then performs a 5-way addition of the four 32-bit intermediate results and the vertically corresponding 32-bit integer element in its 3rd vector operand (serving here as a source operand) and places the result of the 5-way addition in the same place of the 32-bit data element in the 3rd vector operand (now serving as a destination operand). VPDPBUSD replaces an Intel AVX-512 three instruction sequence that accomplishes the same functionality with higher accuracy since Intel AVX-512 saturates the 16-bit intermediate results: VPMADDUBSW + VPMADDWD + VPADDD. See Figure 8-1 below.

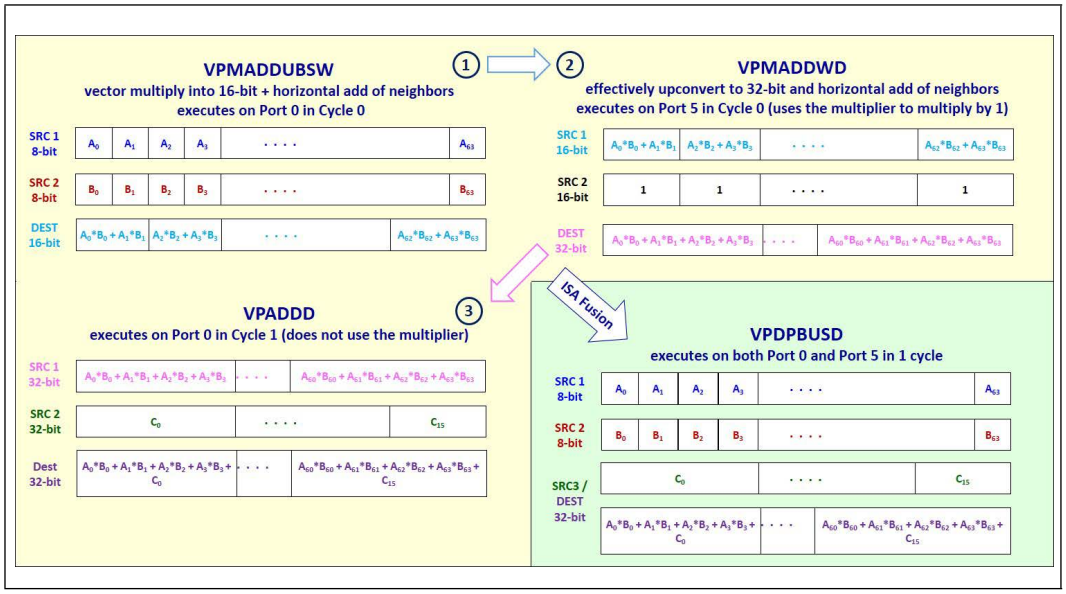


Figure 8-1. VPMADDUBSW + VPMADDWD + VPADDD Fused into VPDPBUSD (3x Peak Ops on Server Architectures, 2x Peak Ops on Client Architectures)

Example 8-1 uses the VPDPBUSD instruction to perform faster matrix multiplication of two byte matrices, SIGNAL and WEIGHT. Assuming the source matrices have dimensions MxK and KxN, respectively, and are given in row-major order, the source matrices in the example have the layouts defined below.

- Matrix signal[K/64][M][64], built out of matrix SIGNAL[M][K] by the following procedure:

  FOR m = 0 … M-1

  FOR k = 0 … K-1

  signal[k/64][m][k%64] = SIGNAL[m][k]

- Matrix weight[K/4][N][4], built out of matrix WEIGHT[K][N] by the following procedure:

  FOR k = 0 … K-1

  FOR n = 0 … N-1

  weight[k/4][n][k%4] = WEIGHT[k][n]

Example 8-1. VPDPBUSD Implementation

Vector Implementation with Pre-Intel® DL Boost Intel® AVX-512)	Intel® DL Boost VPDPBUSD Implementation
// inner loop of unrolled matrix multiply vpbroadcastd zmm31, dword ptr [onew] vpbroadcastd zmm24, [signal] vmovups zmm25, [weight] vmovups zmm26, [weight + 64] vmovups zmm27, [weight + 128] vmovups zmm28, [weight + 192] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm0 , zmm0 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm6 , zmm6 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm12, zmm12, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm18, zmm18, zmm30 vpbroadcastd zmm24, [signal + 64] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm1 , zmm1 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm7 , zmm7 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm13, zmm13, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm19, zmm19, zmm30 vpbroadcastd zmm24, [signal + 128] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm2 , zmm2 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm8 , zmm8 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm14, zmm14, zmm29 vpmaddubsw zmm30, zmm24, zmm28	// inner loop of unrolled matrix multiply vpbroadcastd zmm24, [signal] vpbroadcastd zmm25, [signal + 64] vpbroadcastd zmm26, [signal + 128] vpbroadcastd zmm27, [signal + 192] vmovups zmm28, [weight] vmovups zmm29, [weight + 64] vmovups zmm30, [weight + 128] vmovups zmm31, [weight + 192] vpdpbusd zmm0 , zmm24, zmm28 vpdpbusd zmm6 , zmm24, zmm29 vpdpbusd zmm12, zmm24, zmm30 vpdpbusd zmm18, zmm24, zmm31 vpdpbusd zmm1 , zmm25, zmm28 vpdpbusd zmm7 , zmm25, zmm29 vpdpbusd zmm13, zmm25, zmm30 vpdpbusd zmm19, zmm25, zmm31 vpdpbusd zmm2 , zmm26, zmm28 vpdpbusd zmm8 , zmm26, zmm29 vpdpbusd zmm14, zmm26, zmm30 vpdpbusd zmm20, zmm26, zmm31 vpdpbusd zmm3 , zmm27, zmm28 vpdpbusd zmm9 , zmm27, zmm29 vpdpbusd zmm15, zmm27, zmm30 vpdpbusd zmm21, zmm27, zmm31

Example 8-1. VPDPBUSD Implementation (Contd.)

Vector Implementation with Pre-Intel® DL Boost Intel® AVX-512)	Intel® DL Boost VPDPBUSD Implementation
vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm20, zmm20, zmm30 vpbroadcastd zmm24, [signal + 192] vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm3 , zmm3 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm9 , zmm9 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm15, zmm15, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm21, zmm21, zmm30	
Baseline	Speedup: 2.75x[1]

NOTES:

1. Client architectures based on processors that support Intel® DL Boost, such as processors based on Ice Lake microarchitecture will only see a 2x speedup. This is because VPADDD can exploit the vector SIMD unit on port 5 so the baseline takes 2 cycles per 64 MACs (peak) vs. 1 cycle with Intel® DL Boost.

8.2.2 MULTIPLY & ADD SIGNED WORD INTEGERS (VPDPWSSD INSTRUCTION)

VPDPWSSD is a 16-bit integer multiply-accumulate vector operation into a vector of 32-bit integer accumulators. It accepts two source vector operands and one source/destination vector operand. The two source operands contain 16-bit integer data elements. The source/destination operand contains 32-bit data elements.

If the use of the 8-bit VPDPBUSD instruction introduces an unacceptable loss in inference accuracy, the 16-bit VPDPWSSD instruction can be used instead. Still, it is recommended to revert to FP32 operations in such scenarios until the new BFLOAT16 data type and its associated operations are supported by Intel processors.

No performance gain is expected from the VPDPWSSD instruction on client architectures such as the Ice Lake Client microarchitecture.

8.3 GENERAL OPTIMIZATIONS

8.3.1 MEMORY LAYOUT

Assume the NHWC memory layout is as described in the TensorFlow performance guide. If the inputs are given in native format (either row or column major) the data is converted to an optimized layout at the beginning of the computation with scalar code[1].

1. See the Intel AI & Machine Learning Landing Page for additional details.

8.3.2 QUANTIZATION

Quantization is the process of reducing the size of the data type for activations and weights, typically from floats to int8/uint8[1].

8.3.2.1 Quantization of Weights

Weights are quantized using the quantization factor 127/max_range. This can be done per OFM for increased accuracy. Since weights are known up-front the process can be done offline.

8.3.2.2 Quantization of Activations

The following code snippet shows how to quantize data in scalar or vector fashion, given a quantization factor.

Example 8-2. Quantization of Activations (Scalar)

Quantization of Inputs with a Given Factor (Scalar)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
  {
     int quant_min = 0;
     int quant_max = (1 << bits) - 1;
     #pragma unroll (4)
     for (int i = 0; i < count; i++) {
        int int32_val = offset + (int)round(data[i] * factor);
        int32_val = std::max(std::min(int32_val, quant_max), quant_min);
        u8 quant_val = (u8)int32_val;
       quantized_data[i] = quant_val;
     }
}
```

Example 8-3. Quantization of Activations (Vectorized)

Example 1-1

Quantization of Inputs with a Given Factor (Vectorized)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
  {
     int quant_min = 0;
     int quant_max = (1 << bits) - 1;
     int count_aligned = ALIGN(count, INTR_VECTOR_LENGTH_32_bit);
     __m512i offset_broadcast = _mm512_set1_epi32(offset);
     __m512 factor_broadcast = _mm512_set1_ps(factor);
     __m512i quant_min_broadcast = _mm512_set1_epi32(quant_min);
     __m512i quant_max_broadcast = _mm512_set1_epi32(quant_max);
```

1. Please see: Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) and the AI & Machine Learning Landing page.

Example 1-1 (Contd.)

Quantization of Inputs with a Given Factor (Vectorized)

```
#pragma unroll (4)
      for (int i = 0; i < count_aligned; i += INTR_VECTOR_LENGTH_32_bit) {
      __m512 data_m512 = _mm512_load_ps(&data[i]);
      data_m512 = _mm512_mul_ps(data_m512, factor_broadcast);
      __m512i data_i32 = _mm512_cvt_roundps_epi32 (data_m512
,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
      data_i32 = _mm512_add_epi32(data_i32, offset_broadcast);
      data_i32 = _mm512_max_epi32(data_i32, quant_min_broadcast);
      data_i32 = _mm512_min_epi32(data_i32, quant_max_broadcast);
      __m128i q = _mm512_cvtusepi32_epi8(data_i32);
      _mm_store_si128((__m128i*)(&quantized_data[i]), q);
       }
}
```

8.3.2.3 Quantizing Negative Activations

VPMADDUBSW and VPDPBUSD only support the combination of an unsigned value in the first parameter and a signed value in the second parameter. This means that signed weights (in the second parameter) can be easily supported, but signed activations (in the first parameter) require some manipulations. When there is a possibility of negative activations, e.g., when there is no ReLU before the current layer, we first quantize to the values of -128, 127 and then add 128 to the result to achieve non-negative activations. We compensate for this offset by subtracting 128*(sum of all the weights of the OFM filter) from the OFM bias; see the Intel AI Academy for the full details.

8.3.3 MULTICORE CONSIDERATIONS

8.3.3.1 Large Batch (Throughput Workload)

Computation of large batches can benefit significantly from multicore processing by dividing the work among multiple cores but, due to cache locality, it is best to fully process the same object (image, sentence, etc.) on the same physical core. Furthermore, while the activations are unique per object, the weights can usually be shared. A multithreaded model allows for easy sharing of weights between the cores. Guidelines for processing large batch of input in a multicore system are listed below.

1. Use a thread model to share the weights between the multiple cores. That said, for multi-socket machines there should be a dedicated process per socket/NUMA domain.
2. Define thread affinity and object affinity to fully process a single object in the same physical core, thus keeping the activations in core caches (unless larger than the caches size).
3. Batch objects to a whole multiple of the core count so that the work will be evenly loaded between the cores.
4. Ensure that the sibling thread (logical processor) of every physical core is idle.
5. Consider running a per core mini-batch in BFS mode where, for example, the same layer is executed for all the images assigned to the core before moving to the next layer. This improves weight reuse at the cost of polluting the core caches with multiple activations and (sometimes) improves performance. Mini-batching is quite useful when the sizes of the matrices (tensors) are otherwise very skinny, causing under-utilization of the multiply-accumulate units.

8.3.3.2 Small Batch (Throughput at Latency Workload)

Small batch processing usually has some latency requirements which are not always possible to fulfill with a single core. In such cases it is necessary to split the processing of a single object across multiple cores.

Increasing the number of cores that process a single object often has diminishing returns, so it is best to find the knee point. Sometimes it is possible to slightly increase the batch count and still hit the required latency (e.g., from one to two or three). Once the optimal thread count of a single batch instance is found, multiple instances can be run to fully utilize the system.

8.3.3.3 NUMA

The Cascade Lake Advanced Performance 2-Socket server contains two Cascade Lake Advanced Performance packages where each of the packages is made of two processor dies connected via one Intel® Ultra Path Interconnect (Intel® UPI) link, creating a total of four NUMA domains. In such a setup it is crucial to maintain a separate DL process per NUMA domain/die. This is also the case for previous 2-socket setups of the previous generation product lines with multiple NUMA domains.

8.4 CNNS

8.4.1 CONVOLUTIONAL LAYERS

8.4.1.1 Direct Convolution

To utilize Intel® DL Boost vector operations we strive to reduce the direct convolution operation into a series of matrix additions and multiplications. In the following discussion we denote the matrices replacing the inputs, weights and outputs as A, B and C respectively.

Memory Layout

To present the inputs in matrix form we flatten the spatial dimension to be the rows of A (henceforth called the M dimension), and the channel (IFM) dimension to be the columns of A (henceforth called the K dimension). Similarly the spatial dimension of the outputs becomes the rows of C, and the channel (OFM) dimension becomes the columns of C (henceforth called the N dimension). In Figure 8-2 there are six channels of size 5x5 in the inputs which are transformed by the convolutional layer to four channels of size 3x3.

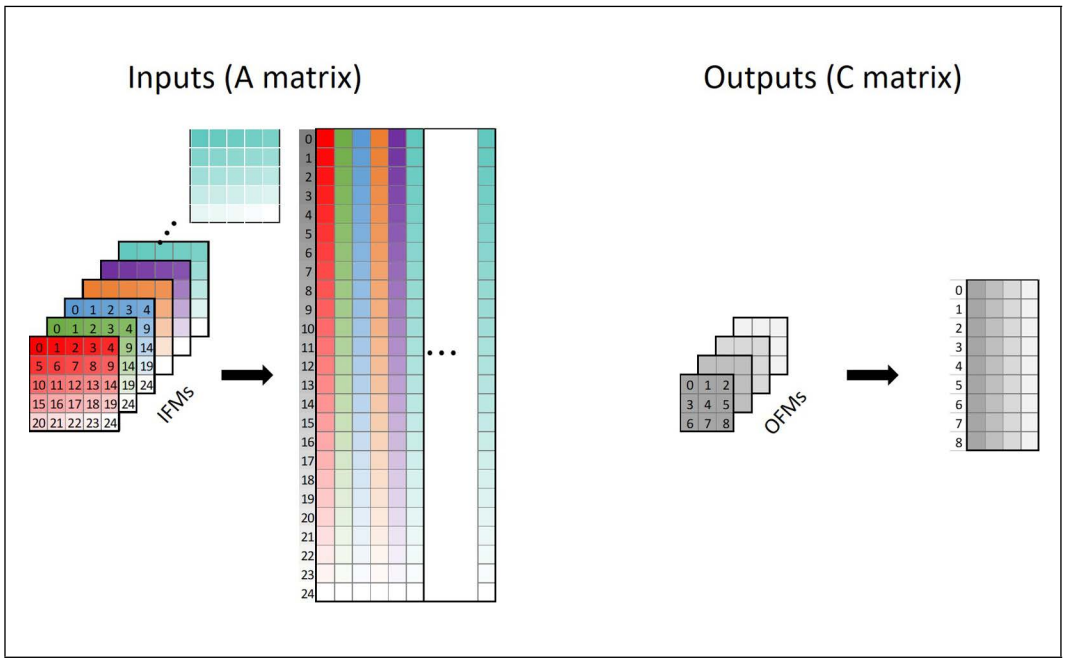


Figure 8-2. Matrix Layout, Inputs and Outputs

Standard 2D convolution uses #OFMs different 3D kernels of size KH x KW x #IFMs, for each target OFM, where KH, KW, #IFMs and #OFMs are the height and width of the convolutional kernel, number of input channels and number of output channels respectively.

The weights are transformed into KH x KW matrices of size #IFMs x #OFMs (see Figure 8-3).

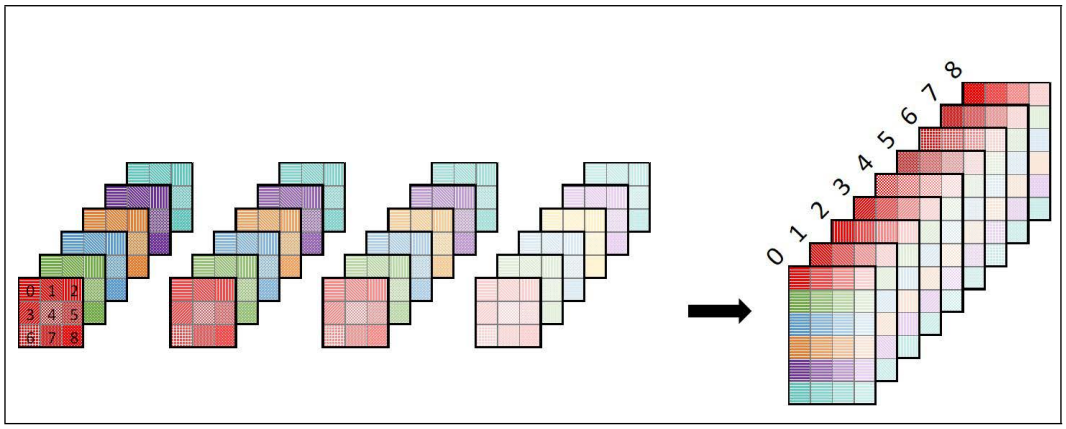


Figure 8-3. Transformed Weights

As a result, the convolution operation (Figure 8-4),

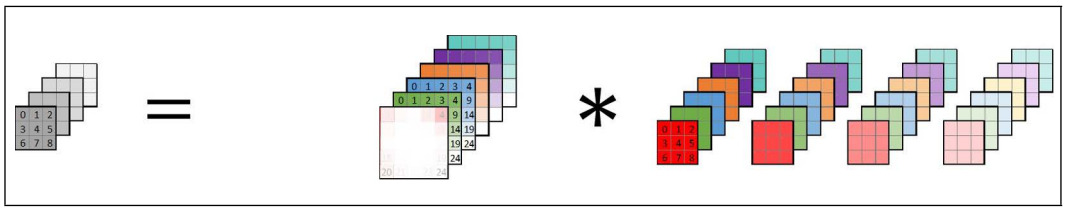


Figure 8-4. Convolution Operation

becomes a series of matrix multiplications and summations (Figure 8-5).

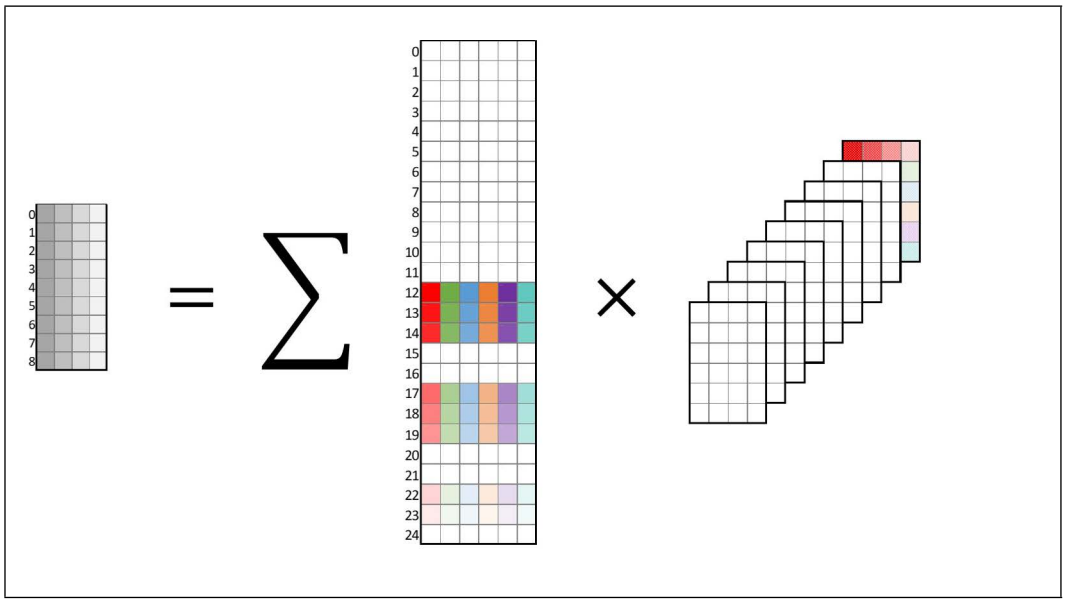


Figure 8-5. Matrix Multiplications and Summations

Matrix Multiplication

Matrix multiplication is performed in a standard manner (see Chapter 17, "Software Optimization for Intel® AVX-512 Instructions").

Blocking

Since the matrices in question are generally large, one needs to traverse the matrices iteratively, while accumulating the results of the multiplication. Hence, one needs to allocate several registers (accumulators) to accumulate the results, and optionally have several registers for temporary caching of the A and B matrices, to enable reuse.

The order in which the matrices are traversed can have a monumental effect on the overall performance. In general, it is preferable to traverse the entire K dimension before moving on to the next element in the M or N dimension. When the K dimension is exhausted, the results in the accumulators are final (see discussion below about partial results). They can be fed to the post-convolution stage (See Post Convolution) and stored to the output location. If, however, the K dimension is not exhausted, before a move in the M or N dimension is initiated, the results in the accumulators are partial, i.e., they are the result of multiplication of some columns of A by some rows of B. These results have to be saved in an auxiliary buffer to free the accumulators for the new chunk of data. When we return to these M, N coordinates, these results have to be loaded from the buffer. Thus we perform additional store(s) and

load(s) when compared to the exhaustive-K scenario. Furthermore, it is generally advisable to limit the advancement in the M or N dimension. Generally speaking, best results were obtained when the Accumulator K cache level of matrix B (Figure 8-6) was in the DCU, and when the accumulative size of the cache blocks (Figure 8-7) was as large as possible while still in MLC. However, there are cases where the best results are achieved when the accumulative size is much larger (even up to 3x of the MLC). These hyper-parameters are usually found by experimentation.

The “exhaustive-K” guideline does not yield optimal results in all cases, and the optimal extent of M,N locality should be determined on a case-by-case basis. We outline a simple yet effective way of structuring the control flow of the convolution process to accommodate the variety of scenarios abundant in modern CNNs.

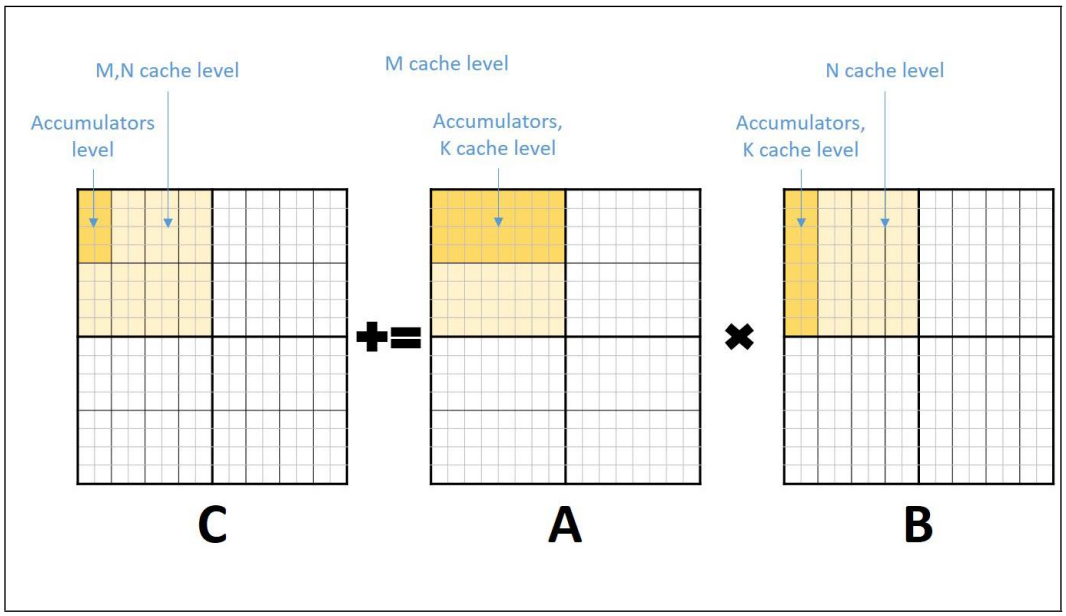


Figure 8-6. 3-Tier Flexible 2D Blocking

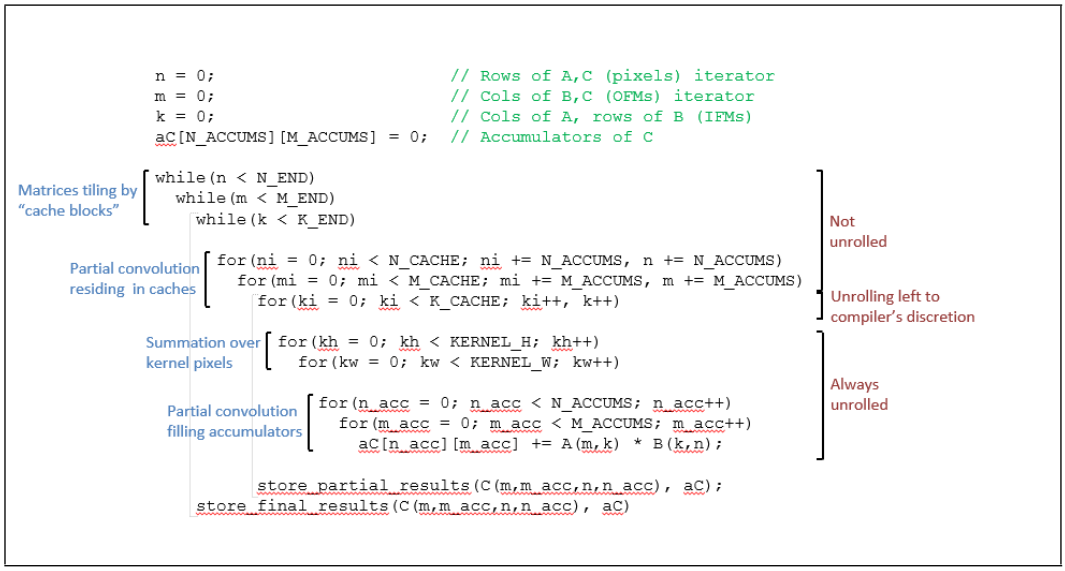


Figure 8-7. 3-Tier Flexible 2D Blocking Loops

Direct Convolution Example

The following code demonstrates the usage of VNNI for direct convolution. The code implements the blocking optimization which is discussed in the previous section. The function direct_conv_opt takes a pointer to a structure with all the parameters of the convolution including two blocking related parameters – the number of accumulators on the M and N dimensions.

'a_buffer_vnni' and 'b_buffer_vnni' point to the A and B matrices, respectively, after they have been reformatted to the VNNI layout.

Blocking is accomplished by allocating a temporary array with the number of accumulators based on the provided parameters (n_accum * m_accum). After the core of the loop populates this array its contents are copied to the correct location in the final C buffer. The two outer loops traverse N and M according to their respective blocking parameters and then the next two loops traverse N and M inside the blocked area. The final three loops traverse the K dimension and then the spatial kernel dimensions. The macros 'a_buffer_vnni_at' and 'b_buffer_vnni_at' are used to access the VNNI buffers in the correct location. We load 4 int8 elements from the A buffer and broadcast them

sixteen times in the ZMM register. Then a ZMM register is loaded from the B buffer and the VNNI instruction is executed on the two registers.

Example 8-4. Direct Convolution

```
struct direct_conv_dims_t {
   // problem params
   int h_dim; // height of inputs
   int w_dim; // width of inputs
   int k_dim; // number of IFMS
   int n_dim; // number of OFMS
   int kh;    // kernel height
   int kw;    // kernel width
   int sh;    // stride height
   int sw;    // stride width

   // blocking params
   int n_accum;
   int m_accum;
};

typedef struct direct_conv_dims_t direct_conv_dims_t;

#define B_MATRIX_BLOCK 4
#define C_MATRIX_BLOCK 16

#define OUT_H(d) (((d)->h_dim - (d)->kh) / (d)->sh + 1)
#define OUT_W(d) (((d)->w_dim - (d)->kw) / (d)->sw + 1)

#define a_buffer_vnni_at(d, a, h, w, k)                                    \
   (a[((h) * (d)->w_dim * (d)->k_dim) + ((w) * (d)->k_dim) + (k)])
#define b_buffer_vnni_at(d, b, k_h, k_w, k_d, n, k_m)                      \
   (b[((k_h) * (d)->kw * ((d)->k_dim / B_MATRIX_BLOCK) * (d)->n_dim *      \
      B_MATRIX_BLOCK) +                                                    \
((k_w) * ((d)->k_dim / B_MATRIX_BLOCK) * (d)->n_dim * B_MATRIX_BLOCK) +    \
     ((k_d) * (d)->n_dim * B_MATRIX_BLOCK) + ((n)*B_MATRIX_BLOCK) + (k_m)])
void direct_conv_opt(const direct_conv_dims_t *dims, const char *a_buffer_vnni,
                const char *b_buffer_vnni, int32_t *c_buffer_vnni)
{
int m_dim = OUT_H(dims) * OUT_W(dims);
    __m512i *cvec =
      _mm_malloc(dims->n_accum * dims->m_accum * sizeof(*cvec), 64);
```

Example 8-4. Direct Convolution (Contd.)

```
    for (int n = 0; n < dims->n_dim; n += C_MATRIX_BLOCK * dims->n_accum) {
     // the '1' represents the size of the VNNI register on the M dimension
    for (int m = 0; m < m_dim; m += 1 * dims->m_accum) {
        for (int i = 0; i < dims->n_accum; i++)
           for (int j = 0; j < dims->m_accum; j++)
              cvec[i * dims->m_accum + j] = _mm512_setzero_epi32();

    for (int ni = 0; ni < dims->n_accum; ni++) {
           int n_final = n + ni * C_MATRIX_BLOCK;
for (int mi = 0; mi < dims->m_accum; mi++) {
              int m_final = m + mi;
              int h = m_final / OUT_W(dims);
              int w = m_final % OUT_W(dims);

              for (int k = 0; k < dims->k_dim; k += B_MATRIX_BLOCK) {
                 for (int k_h = 0; k_h < dims->kh; k_h++) {
                    for (int k_w = 0; k_w < dims->kw; k_w++) {
                       const int32_t *a_addr =
                          (const int32_t *)&a_buffer_vnni_at(
                             dims, a_buffer_vnni, h * dims->sh + k_h,
                             w * dims->sw + k_w, k);
                       const int32_t a = *a_addr;
                       __m512i avec = _mm512_set1_epi32(a);

                       const char *b_addr = &b_buffer_vnni_at(
                          dims, b_buffer_vnni, k_h, k_w,
                          k / B_MATRIX_BLOCK, n_final, 0);
                       __m512i bvec = _mm512_load_si512(b_addr);
cvec[ni * dims->m_accum + mi] =
                          _mm512_dpbusd_epi32(
                             cvec[ni * dims->m_accum + mi], avec,
                             bvec);
                    }
                 }
              }
              int32_t *c_base =
                 &c_buffer_vnni[(m_final * dims->n_dim) + n_final];
              _mm512_store_si512(c_base, cvec[ni * dims->m_accum + mi]);
           }
        }
     }
   }
   _mm_free(cvec);
}
```

In this code we allocate M_ACCUM (4) zmm registers zmm8-zmm11 for IFM values, and N_ACCUM (2) zmm registers zmm12-zm13 for weights.

In the beginning the accumulators must be zeroed out. Then the entire K dimension (#IFMs=32) must be traversed, each iteration operating on 4 consecutive IFMs. The convolution consists from a series of 4-byte broadcasts of IFM data, 64-byte loads of weights data, and multiplication and accumulation operations. Due to the large IFM data overlap between different kh,kw values, the IFM data can be efficiently reused, and the number of data loads significantly lowered.

8.4.1.2 Convolutional Layers with Low OFM Count

Vectorization along the channel dimension works well when there are enough channels (both input and output) to fill up the vector registers, which is usually the case with classification topologies. However, in some cases such as Generative Adversarial Networks (GANs) the end result is an image which means that last convolutional layer has only three channels. In this layer it makes more sense to vectorize along the spatial dimension, which requires a different layout of data. To avoid a large intermediate buffer we re-layout the computation on the fly for one 4x16 block, perform a partial convolution, and throw the block away (this mechanism is limited for 1x1 kernels and assumes the weights have been reordered to match the new layout).

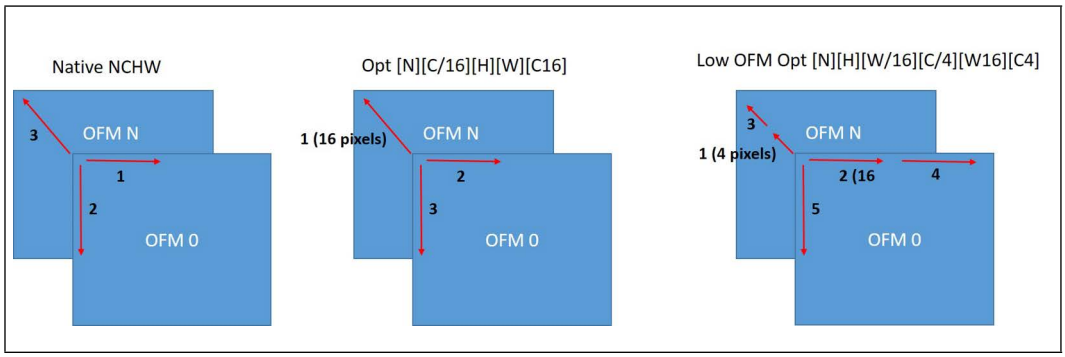


Figure 8-8. Standard vs Optimized vs. Low OFM Optimized Data Layouts[1]

NOTES:

1. The 4x16 blocks of the Low OFM optimization are created on the fly and used only once.

Example 8-5. Convolution for Layers with Low OFM Count

Example 1-2

```
# IFM_W % 16 == 0
# NUM_OFMS = 3
# NUM_IFMS = 64
# dqfs - array of dequantization factors for the down convert

          int src_ifm_size = IFM_H * IFM_W * IFMBlock;
          int ofm_size = IFM_W * IFM_H;

    __m512i gather_indices = _mm512_setr_epi32(0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60);
          __m512 dqf_broadcast[NUM_OFMS];
   #pragma unroll(NUM_OFMS)
   for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
          dqf_broadcast[ofm] = _mm512_set1_ps(dqfs[ofm]);
   }
    for (int h = 0 ; h < IFM_H ; h++) {
          int src_line_offset = h * IFM_W * IFMBlock;
          int w = 0;
           int src_w_offset = src_line_offset;
    for ( ; w < IFM_W ; w += 16) {
          __m512i res_i32[NUM_OFMS] = { 0 };
          __m512i res_i32[NUM_OFMS] = { 0 };
    // Convolve 4x16 OFMs by reorganizing on the fly.
         for (int ifm = 0 ; ifm < NUM_IFMS ; ifm += 4) {
            int src_block_offset = ifm & 0xf;
            int src_ifm_index = ifm >> 4;
            size_t src_ifm_offset = src_w_offset + src_ifm_index * src_ifm_size + src_block_offset;
            __m512i ivec = _mm512_i32gather_epi32(gather_indices, input + src_ifm_offset, 4);
            #pragma unroll(NUM_OFMS)
            for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
               int weight_offset = (ofm * NUM_IFMS + ifm) * 16;
__m512i wvec = _mm512_load_si512(weights_reorged + weight_offset);
               res_i32[ofm] = _mm512_dpbusd_epi32(res_i32[ofm], ivec, wvec);
            }
}
        // Down convert and store results in native layout.
          #pragma unroll(NUM_OFMS)
          for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
            __m512 res_f32 = _mm512_cvtepi32_ps(res_i32[ofm]);
            res_f32 = _mm512_mul_ps(res_f32, dqf_broadcast[ofm]);
size_t output_offset = ofm * ofm_size + h * IFM_W + w;
            _mm512_store_ps(output + output_offset, res_f32);
                }
                src_w_offset += 16 * IFMBlock;
          }
    }
}
```

8.4.2 POST CONVOLUTION

Numerous transformations may be operated on the layer data once the convolution is done. These may include classical post convolution operations such as ReLU, operations that are usually considered a separate layer such as pooling or EltWise, and quantization/dequantization operations. To reduce memory hierarchy thrashing, try to do these steps during the convolution (i.e., fused into the convolution computation). Fusing the quantization step is especially attractive because it gains 4x compute bandwidth and reduces memory bandwidth 4x.

8.4.2.1 Fused Quantization/Dequantization

Section 8.3.2.1 describes how to do offline quantization. Typically however, the dequantization of the current layer and the quantization for the next layer can be fused to the convolution step. The following code describes the basic operation of the post-convolution step which is initiated once the convolution of a block of OFMs was finished. As before, the procedure operates on 16 int8 OFMs belonging to the same pixel. In this example we assume that the dequantization factors (and bias if any) were prepared so that a single factor represents the dequantization of the current layer and then the requantization to the next layer. Generally speaking we try to reduce the number of online computations by representing several multiplication factors, e.g., dequantization, layer constant multiplication value, and quantization, to the next layer as a single factor. In addition, if the original OFMs could have been negative (no ReLu) we follow the procedure of Section 8.3.2.1 and add 127 to all the values.

Example 8-6. Basic PostConv

```
//dest points to a vector of 16 OFMSs belonging to the same pixel.
uint8_t* dest = (uint8_t*) (outputFeatureMaps) + offset;

// in are the 16 accumulators in int32 that we need to operate on.
__m512 resf =_mm512_cvtepi32_ps(in);  // Convert to float

// add bias if there is one and then dequantize + requantize in a single step
if (bias) {
  resf = _mm512_fmadd_ps(resf,
                  _mm512_load_ps(dqfs + OFMChannelOffset),
                  _mm512_load_ps((__m512*) (bias + OFMChannelOffset)));

} else {
  resf = _mm512_mul_ps(resf,
                _mm512_load_ps(dqfs + OFMChannelOffset));
}
```

Example 8-6. Basic PostConv (Contd.)

```
#if RELU
   resf = _mm512_max_ps(resf, broadcast_zero);
#endif

// at this point we are in the uint8 range.
__m512i res = _mm512_cvt_roundps_epi32(resf, _MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
__m128i res8;

#if ELTWISE
    /* fused Eltwise ops */
#else

# if !RELU
     res = _mm512_add_epi32(res, _mm512_set1_epi32(128));
# endif
  res8 = _mm512_cvtusepi32_epi8(res);
#endif  // ELTWISE

#if POOLING
   /* fused pooling ops */
#endif

_mm_store_si128((__m128i*) dest, res8);
```

8.4.2.2 ReLu

ReLu is implemented as a max with a zero register with negligible overhead as it fused with the convolution.

8.4.2.3 EltWise

Element wise operations are usually easier to fuse into the convolution step because they operate directly on the accumulators just before the final result is saved. Note, however, that the quantization factors of the different input layers are usually not the same so the inputs must first be dequantaized to f32, operated on and then quantized again (we show an optimization for this step in the vectorized code).

The following example of the eltwise operation required given the data type of the inputs and outputs. In all the examples it is assumed that the data from the convolutional operation is the INT32 data returned from the VPDPBUSD operation, and that the quantized output must be uint8, even though in some cases the unquantized output could be negative. See Section 8.3.2 to understand how quantization to uint8 works with negative values.

The following optimized code shows eltwise implementations for several eltwise use cases assuming that “dest” points to a vector of 16 OFMs belonging to the same pixel in the output. In principle we need to dequantize the eltwise data and the convolution data, do the addition and then dequantize, as in the following equation.

$$result = \left(eltwise_{f32} \times eltwiseDQfactor + conv_{i32} \times convDQfactor\right) \times NextQfactor$$

However, we can preprocess the factors offline (operations in square brackets) so that we have only two multiplications online.

$$result = \left(eltwise_{f32} + conv_{i32}\left[\frac{convDQfactor}{eltwiseDQfactor}\right]\right) \times ([NextQfactor \times eltwiseDQfactor])$$

Example 8-7. Uint8 Residual Input

Uint8 Residual Input

```
__m128i eltwise_u8 = _mm_load_si128((const __m128i*) (eltwise_data + ew_offset));
__m512i eltwise_i32 = _mm512_cvtepu8_epi32(eltwise_u8);
if (signed_residual) {
   eltwise_i32 = _mm512_sub_epi32(eltwise_i32, broadcast_128);
}
__m512 eltwise_f32 = _mm512_cvtepi32_ps(eltwise_i32);
resf = _mm512_add_ps(eltwise_f32,resf); /* add with conv results */

/* dequantization and then requantization to next layer in one op */
resf = _mm512_mul_ps(resf, broadcast_fused_eltwise_out_qfactor);
if (relu)
   resf = _mm512_max_ps(resf, broadcast_zero);
__m512i data_i32 = _mm512_cvt_roundps_epi32(resf,
                                (_MM_FROUND_TO_NEAREST_INT|
                                 _MM_FROUND_NO_EXC));
res8 = _mm512_cvtusepi32_epi8(data_i32);

if (!relu) {
   res8 = _mm_add_epi8(res8, _mm_set1_epi8(-128)); //hack to add 128
}
```

8.4.2.4 Pooling

Pooling layers may not be simple to fuse to the convolution step but in some cases the fusion is easy. The average pooling of the last convolutional layer of Inception ResNet 50 for example amounts to averaging all the 8x8 pixels of every OFM channel into a single value, thus emitting a single value per OFM. Such an operation is easy to fuse because it behaves the same for every pixel.

Example 8-8. 8x8 Average Pooling with Stride 1 of 8x8 Layers

8x8 Average Pooling with Stride 1 of 8x8 Layers

```
__m512 pool_factor  = _mm512_set1_ps((float)1.0/64);

// resf is the 16 float values as computed in Basic PostConv code sample

resf = _mm512_mul_ps(resf, pool_factor); // divide by 64

// The pool offset depends only on the current OFM (OFMItr).
int pool_offset = (BlockOffsetOFM + OFMItr);
float *pool_dest = (float *) (outputFeatureMaps) + pool_offset;
__m512 prev_val = _mm512_load_ps((const __m512 *) (pool_dest));
```

Example 8-8. 8x8 Average Pooling with Stride 1 of 8x8 Layers (Contd.)

8x8 Average Pooling with Stride 1 of 8x8 Layers

```
__m512 res_tmp_ps = _mm512_add_ps(resf, prev_val);
_mm512_store_ps((__m512 *) pool_dest, res_tmp_ps);
```

The following unfused vectorized code can be used to do max and average pooling. In the example below the pooling step can also adjust the input quantization range to the output quantization range. This is usually necessary before a concat layer, which is implemented as a No-Op, which means that the output quantization range of all the concatenated layers must be the same.

Example 8-9. Unfused Vectorized Pooling

Unfused Vectorized Pooling

```
// We concurrency pool 16 IFMs before moving to the next set of IFMs
for (int ifm = 0; ifm < no_ifm; ifm+=16) {
   // Find the location of the block in the input and in the output that we are pooling
   int block_idx = (ifm >> 4) ;
   size_t block_offset = (spatial_size_in * block_idx) << 4;
   size_t block_offset_out = (spatial_size_out * block_idx) << 4;

   for (int y = -pad_h_; y < top_y + pad_h_; y++) {
      int y_offset_out = (top_x * 16 * (y + pad_h_));
      for (int x = -pad_w_; x < top_x + pad_w_; x++) {
         __m256i res_pixel = _mm256_set1_epi16(0); //should be u_int but 0 is 0
         for (int i = 0; i < kernel_w; i++) {
            int y_offset_in = (bottom_x * (i + (y * stride)))<<4;
for (int j = 0; j < kernel_h; j++) {
               int x_offset = (j + (x * stride)) << 4;

               // skip pixels that are inside the pad
               if (pad && ((j + (x * stride)) < 0 || (i + (y * stride)) < 0))
                  continue;

               //load pixel data
__m128i data_px = _mm_load_si128((const __m128i *) (bottom_data+ ifm_image_offset + block_offset +
y_ofset_in + x_ofsset ));
               // convert to u16 for average.
               __m256i data_px_u16 = _mm256_cvtepu8_epi16(data_px);
               if (MAX) {
                  res_pixel = _mm256_max_epu16(res_pixel,data_px_u16);
               } else if (AVERAGE) {
                  res_pixel = _mm256_adds_epu16(res_pixel,data_px_u16);
               }
         } // kernel_h
         } // kernl_w
// done with the input data but there may be some adjustments necessary.
         int x_offset_out = (x + pad_w_) << 4;
         if (SHOOUL_ADJUST_QUANTIZATION_RANGE) {  // typically before a no-op concat
            float factor = layer_param().quantization_param().bottom_range()
                     / this->layer_param().quantization_param().top_range();
```

Example 8-9. Unfused Vectorized Pooling (Contd.)

Unfused Vectorized Pooling

```
            __m512 broadcast_factor = _mm512_set1_ps(factor);

            __m512i res_pixel_i32 = _mm512_cvtepu16_epi32(res_pixel);
            __m512 res_pixel_f32 = _mm512_cvtepi32_ps(res_pixel_i32);
            res_pixel_f32 = _mm512_mul_ps(res_pixel_f32,broadcast_factor);

            __m512i data_i32 = _mm512_cvt_roundps_epi32 (res_pixel_f32
,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
            res_pixel= _mm512_cvtusepi32_epi16(data_i32);

         }
         if (AVERAGE) {
            uint8_t kernel_size_u8 = kernel_h_ * kernel_w_;
            __m256i broadcast_kernel_size = _mm256_set1_epi16(kernel_size_u8);

            res_pixel=_mm256_div_epu16(res_pixel,broadcast_kernel_size);
         }
         // compute final offset and save
         uint8_t * total_offset =
               top_data + output_image_offset + layer_offset + block_offset_out +
                     y_ofsset_out + x_ofsset_out ;//+ vect_idx;
         _mm_store_si128((__m128i*) total_offset, _mm256_cvtusepi16_epi8(res_pixel));

      }
   }
}
```

8.4.2.5 Pixel Shuffler

The SRGAN topology includes a layer that reshapes the inputs as follows.

1. The width and height of the features maps are doubled.
2. The number of output feature maps is divided by four.

Every 2x2 quad in the output feature map is populated by four pixels from the same spatial dimensions in the input which satisfy the condition that (c mod K/4) is the same where c is the input channel and K is the number of input feature maps (reference paper SRGAN).

Example 8-10. Caffe Scalar Code for Pixel Shuffler

```
void pixel_shuffler(const vector<int>& bottom_shape, const vector<int>& top_shape, const pstype*
bottom_data,
                pstype* top_data)
{
const int N = bottom_shape[0];
assert(N == top_shape[0]);
const int bc = bottom_shape[1];
const int bh = bottom_shape[2];
const int bw = bottom_shape[3];
const int tc = top_shape[1];
```

Example 8-10. Caffe Scalar Code for Pixel Shuffler (Contd.)

```
const int th = top_shape[2];
const int tw = top_shape[3];
const int r = th / bh;
int bottom_ch_size = bw * bh;
int top_ch_size = tw * th;
pstype* cur_channel = NULL;
for(int n = 0; n < N; n++){
  const pstype* src = bottom_data + n * bc * bottom_ch_size;
  pstype* dst = top_data + n * tc * top_ch_size;
  for(int c = 0; c < tc; c++){
     cur_channel = dst + c * top_ch_size;
     for(int h = 0; h < bh; h++){
        for(int w = 0; w < bw; w++){
           int bottom_offset = h * bw + w;
           int bottom_index = c * bottom_ch_size + bottom_offset;
           int top_index = h * r * tw + w * r;
           cur_channel[top_index] = src[bottom_index]; // top left
           bottom_index = (c + tc) * bottom_ch_size + bottom_offset;
           top_index = h * r * tw + w * r + 1;
           cur_channel[top_index] = src[bottom_index]; // top right
           bottom_index = (c + 2 * tc) * bottom_ch_size + bottom_offset;
           top_index = (h * r + 1) * tw + w * r;
           cur_channel[top_index] = src[bottom_index]; // bottom left
           bottom_index = (c + 3 * tc) * bottom_ch_size + bottom_offset;
           top_index = (h *r + 1) * tw + w * r + 1;
           cur_channel[top_index] = src[bottom_index]; // bottom right
        }
     }
  }
}
}
```

Because of the memory layout of the vectorized directConv, it is easy to fuse the pixel shuffler layer to the convolution. The only change that is required is to save the result of the convolution in the correct place in the output.

Example 8-11. Computing Output Offset for Fused Pixel Shuffler

```
// base_ofm - output target location (base_ofm % 16 == 0)
// SubTileX - X location in quad (0 or 1)
// SubTileY - Y location in quad (0 or 1)
// ConvOutputX - X position in the output of the convolution
// ConvOutputY - Y position in the output of the convolution

int PostPSNoOutMs =(NoOutFMs / 4);
int PostPSOptOfmIndex = (base_ofm % PostPSNoOutMs) / 16;
int QuarterIndex = base_ofm / PostPSNoOutMs;
int SubTileX = QuarterIndex & 0x1;
int SubTileY = (QuarterIndex & 0x2) >> 1;
int PostPSX = ConvOutputX * 2 + SubTileX;
int PostPSY = ConvOutputY * 2 + SubTileY;
size_t offset = (OFM_H * OFM_W * PostPSOptOfmIndex + PostPSY * OFM_W + PostPSX) * 16;
```

8.5 LSTM NETWORKS

Long short-term memory (LSTM) units are used to create Recurrent Neural Networks (RNNs) for tasks such as speech and text translation. The fundamental computation of an LSTM cell is matrix multiplication (GEMM), not direct convolution, as in CNNs.

8.5.1 FUSED LSTM EMBEDDING

The LSTM cell starts by multiplying the input data by the input kernel where the input data (a.k.a embedding) are 512 elements for every dictionary word and the input kernel is known offline and never changes. Finally, each embedding vector is multiplied by the same matrix. It has been suggested[1] to multiply each vector by the kernel offline and save the result, and at runtime lookup the GEMM result by the word index and copy into the accumulator area of the cell. This optimization may give an approximate 20% performance boost.

8.5.2 FUSED POST GEMM

Many variants of existing LSTM cells contain transcendental operations such as sigmoid and hyperbolic tangent as activation functions.

$$sigmoid(x) = \frac{1}{e^{-x}+1}$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Implementing the activation part as full precision scalar or SMVL-based vectorized code may be slow. The alternative is to use approximations which provide good performance. One of the approaches for approximating transcendental functions is to use piece-wise polynomial approximations.

Example 8-12. Sigmoid Approximation with Minimax Polynomials

```
// clang-format off

//coefficients of second order minimax polynomial for sigmoid approximation
__declspec( align(64) ) const float sigmoid_poly2_coeffs[3][16] = {
{
  0.559464f, 0.702775f, 0.869169f, 0.968227f, 0.996341f, 0.999999f, 0.499999f, 0.499973f,
  0.499924f, 0.499791f, 0.499419f, 0.498471f, 0.496119f, 0.491507f, 0.486298f, 0.495135f,
   },
   {
  0.22038f, 0.123901f, 0.042184f, 0.00779019f, 0.000651011f, 1.12481e-7f, 0.250103f, 0.250739f,
  0.251492f, 0.252905f, 0.255751f, 0.260808f,   0.269823f, 0.282225f, 0.292552f, 0.281425f,
   },
{
  -0.0298035f, -0.0135297f, -0.00347128f, -0.000483042f, -0.0000289636f, -2.57464e-9f, -0.00292674f,
  -0.00680854f, -0.00968539f, -0.0134544f, -0.0188995f, -0.0256562f, -0.0343136f, -0.0426696f, -0.0478004f,
  -0.0443023f,
```

1. Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard M Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In ACL (1). Citeseer, pages 1370-1380.

Example 8-12. Sigmoid Approximation with Minimax Polynomials (Contd.)

```
  },
};


// clang-format on

inline void sigmoid_poly_2(const __m512& arg, __m512& func)
{
   // Load polynomial coefficients into registers (one time operation)
   const __m512 sigmoid_coeff0 = _mm512_load_ps( sigmoid_poly2_coeffs[0] );
   const __m512 sigmoid_coeff1 = _mm512_load_ps( sigmoid_poly2_coeffs[1] );
   const __m512 sigmoid_coeff2 = _mm512_load_ps( sigmoid_poly2_coeffs[2] );
// Extract signs of args
   const __m512 ps_sign_filter = _mm512_castsi512_ps(_mm512_set1_epi32( 0x7FFFFFFF ));

   __mmask16 signs = _mm512_movepi32_mask(_mm512_castps_si512(arg));
   __m512 abs_arg = _mm512_and_ps(arg, ps_sign_filter);

// Compute approximation intervals out of args' exponent and MSB and
   // restrict number of intervals to 16
   const __m512i lut_low = _mm512_set1_epi32( 246 );
   const __m512i lut_high = _mm512_set1_epi32( 261 );

__m512i indices = _mm512_srli_epi32(_mm512_castps_si512(abs_arg), 22);
   indices = _mm512_max_epi32(indices, lut_low);
   indices = _mm512_min_epi32(indices, lut_high);

/*
      * Approximate
      */
      __m512 func_p0 = _mm512_permutexvar_ps(indices, sigmoid_coeff0);
      __m512 func_p1 = _mm512_permutexvar_ps(indices, sigmoid_coeff1);
      __m512 func_p2 = _mm512_permutexvar_ps(indices, sigmoid_coeff2);
func = _mm512_fmadd_ps(abs_arg, func_p2, func_p1);
   func = _mm512_fmadd_ps(abs_arg, func, func_p0);

   // Account for args' sign
  const __m512 ps_ones = _mm512_set1_ps( 1.0 );
   func = _mm512_mask_sub_ps(func, signs, ps_ones, func);
}
```

While the minimax polynomial approximation may show the best accuracy on a layer-by-layer basis, the end-to-end accuracy may suffer in some topologies (NMT notably). In such cases a different approach might be better. The following approximation uses the fact that

$$e^x = 2^{x \log_2 e} = 2^{n+y}$$

Where

$$n = \text{round}(x \log_2 e)$$

$$y = x \log_2 e - n$$

2^n is computed by the scalef instruction

$$2^n = \text{scalef}(x \log_2 e)$$

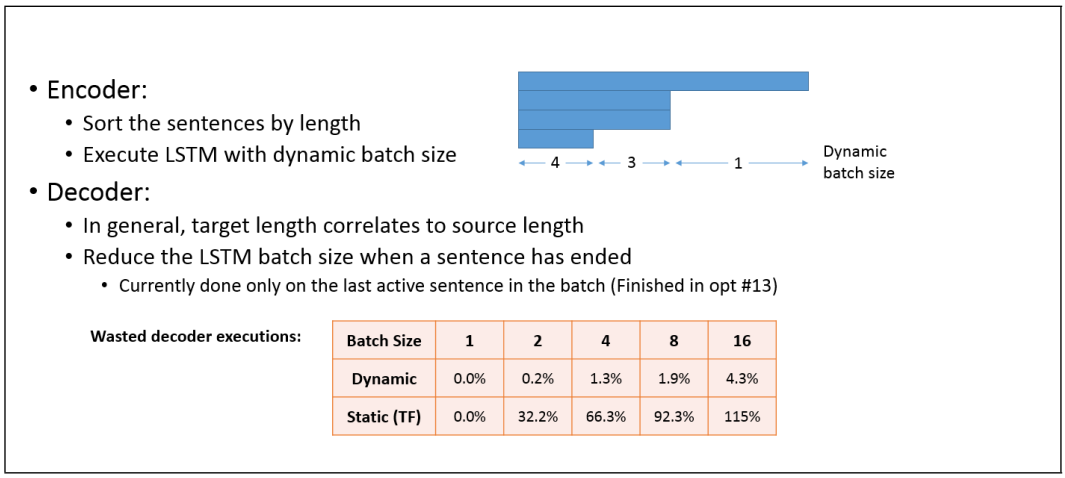
2^y can be approximated sufficiently well by a Taylor Polynomial of degree 2.

Example 8-13. Sigmoid Approximation with Scalef

```
const __m512 ps_ones = _mm512_set1_ps( 1.0 );
const __m512 half = _mm512_set1_ps( 0.5f );
const __m512 minus_log2_e = _mm512_set1_ps( -1.442695f );
const __m512 ln2sq_over_2 = _mm512_set1_ps( 0.240226507f );
const __m512 ln2__ln2sq_over_2 = _mm512_set1_ps( 0.452920674f );
const __m512 one__ln2sq_over_8 = _mm512_set1_ps( 0.713483036f );

inline void sigmoid_scalef(const __m512& arg, __m512& func)
{
   __m512 x = _mm512_fmadd_ps(arg, minus_log2_e, half);
   __m512 y = _mm512_reduce_ps(x, 1);
   __m512 _2y = _mm512_fmadd_ps(_mm512_fmadd_ps(y, ln2sq_over_2, ln2__ln2sq_over_2), y,
one__ln2sq_over_8);
   __m512 exp = _mm512_scalef_ps(_2y, x);
   func = _mm512_rcp14_ps(_mm512_add_ps(exp, ps_ones));
}
```

8.5.3 DYNAMIC BATCH SIZE



Batch Size	1	2	4	8	16
Dynamic	0.0%	0.2%	1.3%	1.9%	4.3%
Static (TF)	0.0%	32.2%	66.3%	92.3%	115%

Figure 8-9. Dynamic Batch Size[1]

NOTES:

1. NMT can gain significantly by adapting the computation in each iteration to the number of sentences that are still active.

Different RNN objects, e.g., sentences, can require very different computation effort, e.g., short sentence vs long sentence. When batching multiple objects together, it is important take this fact into consideration to avoid unnecessary computations. In NMT for example (see Figure 8-9), if we ensure sentences are ordered by length it is easy to adapt each iteration to the actual number of active sentences.

8.5.4 NMT EXAMPLE: BEAM SEARCH DECODER GET TOP K

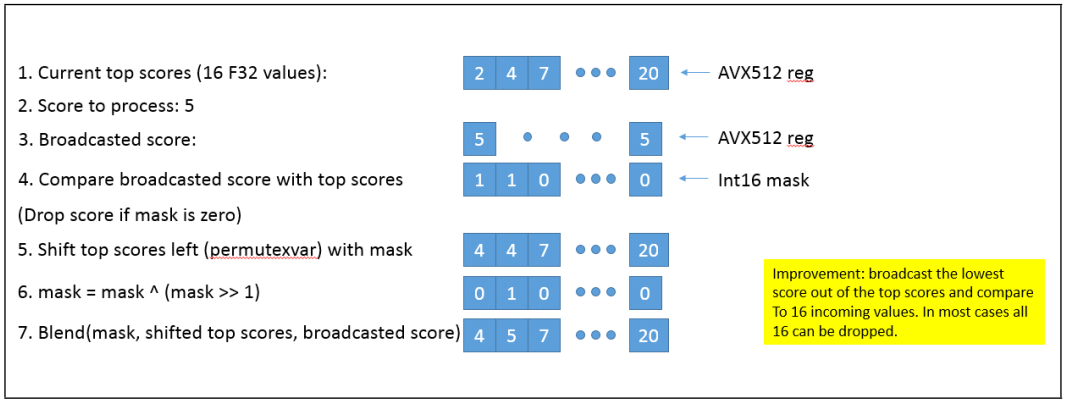


Figure 8-10. Find Top 16 Values in Some Input

In Neural Machine Translation (NMT), a significant amount of time is spent searching for the current top BEAM_WIDTH attention scores out of BEAM_WIDTH*VOCAB_SIZE values, which could be very large. Please see the following for more information:

- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation (seq2seq) Tutorial (GitHub: read only)

We suggest using the following algorithm to optimize this step. The crux of the process is that a new value can be concurrently compared against all the current top values in one op (see Figure 8-10, line 4). Note that we keep the top scores sorted so the mask returned by the op must consist of a sequence of ones followed by a sequence zeros (1*0*).

Example 8-14. Pseudocode for Finding Top K

Pseudocode for Finding Top K

```
// ZMM0 - best scores, initialized to -MAX_FLOAT
// ZMM1 - indices of best scores
// ZMM4 - index the current score

index = 0
pxor ZMM4
while index < MAX_DATA
  vbroadcastss ZMM2, array[index]
  VPCMPPS K1,ZMM0,ZMM2, _CMP_LT_OQ
  KTESTW K1,K1
  JZ …  // K1 == 0 so we can just put new score first
  //if K1!=0
  VPERMPS ZMM0(k1),ZMM0
  VPERMPS ZMM1(k1),ZMM0
  KSHIFT k2,k1,1
  KXOR k3,k2,k1
  VPBLENDMPS k3, ZMM0,ZMM2
VPBLENDMD k3, ZMM1,ZMM4
  VPADD ZMM4, 1
  add index, 1
```

CHAPTER 9
OPTIMIZING CACHE USAGE

While processor speed has increased, memory access speed has increased at a slower pace. The resulting disparity has made it important to tune applications in one of two ways:

1. A majority of data accesses are fulfilled from processor caches.
2. Effectively masking memory latency to utilize peak memory bandwidth as much as possible.

Hardware prefetching mechanisms are enhancements in microarchitecture to facilitate the latter aspect, and will be most effective when combined with software tuning. The performance of most applications can be considerably improved if the data required can be fetched from the processor caches or if memory traffic can take advantage of hardware prefetching effectively.

Standard techniques to bring data into the processor before it is needed involve additional programming which can be difficult to implement and may require special steps to prevent performance degradation. Streaming SIMD Extensions addressed this issue by providing various prefetch instructions.

Streaming SIMD Extensions introduced the various non-temporal store instructions. SSE2 extends this support to new data types and also introduce non-temporal store support for the 32-bit integer registers.

This chapter focuses on:

- Hardware Prefetch Mechanism, Software Prefetch and Cacheability Instructions — Discusses microarchitectural feature and instructions that allow you to affect data caching in an application.
- Memory Optimization Using Hardware Prefetching, Software Prefetch and Cacheability Instructions
  - — Discusses techniques for implementing memory optimizations using the above instructions.
- Using deterministic cache parameters to manage cache hierarchy.

9.1 GENERAL PREFETCH CODING GUIDELINES

The following guidelines will help you to reduce memory traffic and utilize peak memory system bandwidth more effectively when large amounts of data movement must originate from the memory system:

- Take advantage of the hardware prefetcher’s ability to prefetch data that are accessed in linear patterns, in either a forward or backward direction.
- Take advantage of the hardware prefetcher’s ability to prefetch data that are accessed in a regular pattern with access strides that are substantially smaller than half of the trigger distance of the hardware prefetch.
- Facilitate compiler optimization by:
  - — Minimize use of global variables and pointers.
  - — Minimize use of complex control flow.
  - — Use the const modifier, avoid register modifier.
  - — Choose data types carefully (see below) and avoid type casting.
- Use cache blocking techniques (for example, strip mining) as follows:
  - — Improve cache hit rate by using cache blocking techniques such as strip-mining (one dimensional arrays) or loop blocking (two dimensional arrays).
  - — Explore using hardware prefetching mechanism if your data access pattern has sufficient regularity to allow alternate sequencing of data accesses (for example: tiling) for improved spatial locality. Otherwise use PREFETCHNTA.
- Balance single-pass versus multi-pass execution:

  - Single-pass, or unlayered execution passes a single data element through an entire computation pipeline.
  - Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements before passing the entire batch on to the next stage.
  - If your algorithm is single-pass use PREFETCHNTA. If your algorithm is multi-pass use PREFETCHT0.
- Resolve memory bank conflict issues. Minimize memory bank conflicts by applying array grouping to group contiguously used data together or by allocating data within 4-KByte memory pages.
- Resolve cache management issues. Minimize the disturbance of temporal data held within processor's caches by using streaming store instructions.
- Optimize software prefetch scheduling distance:
  - Far ahead enough to allow interim computations to overlap memory access time.
  - Near enough that prefetched data is not replaced from the data cache.
- Use software prefetch concatenation. Arrange prefetches to avoid unnecessary prefetches at the end of an inner loop and to prefetch the first few iterations of the inner loop inside the next outer loop.
- Minimize the number of software prefetches. Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources; excessive usage of prefetches can adversely impact application performance.
- Interleave prefetches with computation instructions. For best performance, software prefetch instructions must be interspersed with computational instructions in the instruction sequence (rather than clustered together).

9.2 PREFETCH AND CACHEABILITY INSTRUCTIONS

The PREFETCH instruction, inserted by the programmers or compilers, accesses a cache line prior to the data actually being needed. This hides the latency for data access in the time required to process data already resident in the cache.

Many algorithms can provide information in advance about the data that is to be required. In cases where memory accesses are in long, regular data patterns; the automatic hardware prefetcher should be favored over software prefetches.

The cacheability control instructions allow you to control data caching strategy to increase cache efficiency and minimize cache pollution.

Data reference patterns can be classified as follows:

- **Temporal:** Data will be used again soon.
- **Spatial:** Data will be used in adjacent locations (for example, on the same cache line).
- **Non-temporal:** Data which is referenced once and not reused in the immediate future (for example, for some multimedia data types, as the vertex buffer in a 3D graphics application).

These data characteristics are used in the discussions that follow.

9.3 PREFETCH

This section discusses the mechanics of the software PREFETCH instructions. In general, software prefetch instructions should be used to supplement the practice of tuning an access pattern to suit the automatic hardware prefetch mechanism.

9.3.1 SOFTWARE DATA PREFETCH

The PREFETCH instruction can hide the latency of data access in performance-critical sections of application code by allowing data to be fetched in advance of actual usage. PREFETCH instructions do not change the user-visible semantics of a program, although they may impact program performance. PREFETCH merely provides a hint to the hardware and generally does not generate exceptions or faults.

PREFETCH loads either non-temporal data or temporal data in the specified cache level. This data access type and the cache level are specified as a hint. Depending on the implementation, the instruction fetches 32 or more aligned bytes (including the specified address byte) into the instruction-specified cache levels.

PREFETCH is implementation-specific; applications need to be tuned to each implementation to maximize performance.

NOTE

Using the PREFETCH instruction is recommended only if data does not fit in cache. Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance penalty. For example specifying a NULL pointer (0L) as address for a prefetch can cause long delays.

PREFETCH provides a hint to the hardware; it does not generate exceptions or faults except for a few special cases (see Section 9.3.3). However, excessive use of PREFETCH instructions may waste memory bandwidth and result in a performance penalty due to resource constraints.

Nevertheless, PREFETCH can lessen the overhead of memory transactions by preventing cache pollution and by using caches and memory efficiently. This is particularly important for applications that share critical system resources, such as the memory bus. See an example in Section 9.6.2.1

PREFETCH is mainly designed to improve application performance by hiding memory latency in the background. If segments of an application access data in a predictable manner (for example, using arrays with known strides), they are good candidates for using PREFETCH to improve performance.

Use the PREFETCH instructions in:

- Predictable memory access patterns.
- Time-consuming innermost loops.
- Locations where the execution pipeline may stall if data is unavailable.

9.3.2 PREFETCH INSTRUCTIONS

Streaming SIMD Extensions include four PREFETCH instruction variants; one non-temporal and three temporal. They correspond to two types of operations, temporal and non-temporal.

Additionally, the PREFETCHW instruction is a hint to fetch data closer to the processor and invalidates any other cached copy in anticipation of a write.

Software prefetch instructions will fetch a 64 byte line of data from memory that contains the byte specified with the source operand. Software prefetch instructions always fetch 64 bytes of data, and because the instructions operate on bytes, can never be split across cache-lines. Thus, a single software prefetch cannot be used to fetch 128 bytes of data.

NOTE

At the time of PREFETCH, if data is already found in a cache level that is closer to the processor than the cache level specified by the instruction, no data movement occurs.

The implementation details of the prefetch hint instructions vary across different microarchitectures. A summary is given in the tables below.

Table 9-1. Implementation Details of Prefetch Hint Instructions for Intel® Core™ Duo, Intel® Core™ 2, and Intel® Atom® Processors

Instruction	Fill Cache? L1	Fill Cache? L2
PrefetchT0	Yes	Yes
PrefetchT1	No	Yes
PrefetchT2	No	Yes
PrefetchNTA	Yes	No
PrefetchW[1]	Yes	Yes

NOTES:

1. PrefetchW is only available on Intel Atom processors; not Intel Core duo or Intel Core 2 processors.

Table 9-2. Details of Prefetch Hint Instructions for Processors Based on Skylake Microarchitectures

Instruction	Fill Cache? L1	Fill Cache? L2	Fill Cache? L3
PrefetchT0	Yes	Yes	No
PrefetchT1	No	Yes	No
PrefetchT2	No	Yes	No
PrefetchNTA	Yes	No	No
PrefetchW[4]	Yes	Yes	No

Table 9-3. Implementation Details of Prefetch Hint Instructions for Intel® Xeon® Scalable Family of Processors

Instruction	Fill Cache? L1	Fill Cache? L2	Fill Cache? L3	Fill Snoop Filter?
PrefetchT0	Yes	Yes	Yes	Yes
PrefetchT1[1]	No	Yes	Yes	Yes
PrefetchT2[2]	No	Yes	Yes	Yes
PrefetchNTA	Yes	No	Yes[2]	Yes[3]
PrefetchW[3]	Yes	Yes	Yes	Yes

NOTES:

1. There is no implementation difference between PrefetchT1/T2 on any microarchitecture.
2. For PrefetchNTA, the fill into the L3 cache or Snoop Filter may not be placed into the Most Recently Used positioned and may be chosen for replacement faster than a regular cache fill.
3. PrefetchW is only available on processors based on Broadwell/Skylake microarchitecture; it is unavailable on processors based on Haswell microarchitecture or earlier microarchitectures.

9.3.3 PREFETCH AND LOAD INSTRUCTIONS

Most of the recent generations of microarchitectures have decoupled execution and memory pipelines. This allows instructions to be executed independently with memory accesses if there are no data and resource dependencies. Programs or compilers can use dummy load instructions to imitate PREFETCH functionality, but preloading is not completely equivalent to using PREFETCH instructions. PREFETCH provides greater performance than preloading.

PREFETCH can provide greater performance than preloading because:

- Has no destination register, it only updates cache lines.
- Does not stall the normal instruction retirement.
- Does not affect the functional behavior of the program.
- Has no cache line split accesses.
- Does not cause exceptions except when the LOCK prefix is used. The LOCK prefix is not a valid prefix for use with PREFETCH.
- Does not complete its own execution if that would cause a fault.

The advantages of PREFETCH over preloading instructions are processor specific. This may change in the future.

There are cases where a PREFETCH will not perform the data prefetch. These include:

- In older microarchitectures, PREFETCH causing a Data Translation Lookaside Buffer (DTLB) miss would be dropped. In processors based on Nehalem, Westmere, Sandy Bridge, and newer microarchitectures, Intel Core 2 processors, and Intel Atom processors, PREFETCH causing a DTLB miss can be fetched across a page boundary.
- An access to the specified address that causes a fault/exception.
- If the memory subsystem runs out of request buffers between the first-level cache and the second-level cache.
- PREFETCH targets an uncacheable memory region (for example, USWC and UC).
- The LOCK prefix is used. This causes an invalid opcode exception.

9.4 CACHEABILITY CONTROL

This section covers the mechanics of cacheability control instructions.

9.4.1 THE NON-TEMPORAL STORE INSTRUCTIONS

This section describes the behavior of streaming stores and reiterates some of the information presented in the previous section.

In Streaming SIMD Extensions, the MOVNTPS, MOVNTPD, MOVNTQ, MOVNTDQ, MOVNTI, MASKMOVQ and MASKMOVDQU instructions are streaming, non-temporal stores. Memory characteristics and ordering are similar to the Write-Combining (WC) memory type:

- **Write-Combining:** Successive writes to the same cache line are combined.
- **Write-Collapsing:** Successive writes to the same byte(s) result in only the last write being visible.
- **Weakly-Ordered:** No ordering is preserved between WC stores or between WC stores and other loads or stores.
- **Uncacheable and Not Write-Allocating:** Stored data is written around the cache and will not generate a read-for-ownership bus request for the corresponding cache line.

9.4.1.1 Fencing

Because streaming stores are weakly ordered, a fencing operation is required to ensure that the stored data is flushed from the processor to memory. Failure to use an appropriate fence may result in data being "trapped" within the processor and will prevent visibility of this data by other processors or system agents.

WC stores require software to ensure coherence of data by performing the fencing operation. See Section 9.4.5

9.4.1.2 Streaming Non-temporal Stores

Streaming stores can improve performance by:

- Increasing store bandwidth if the 64 bytes that fit within a cache line are written consecutively (since they do not require read-for-ownership bus requests and 64 bytes are combined into a single bus write transaction).
- Reducing disturbance of frequently used cached (temporal) data (since they write around the processor caches).

Streaming stores allow cross-aliasing of memory types for a given memory region. For instance, a region may be mapped as write-back (WB) using page attribute tables (PAT) or memory type range registers (MTRRs) and yet is written using a streaming store.

9.4.1.3 Memory Type and Non-temporal Stores

Memory type can take precedence over a non-temporal hint, leading to the following considerations:

- If the programmer specifies a non-temporal store to strongly-ordered uncacheable memory (for example, Uncacheable (UC) or Write-Protect (WP) memory types), then the store behaves like an uncacheable store. The non-temporal hint is ignored and the memory type for the region is retained.
- If the programmer specifies the weakly-ordered uncacheable memory type of Write-Combining (WC), then the non-temporal store and the region have the same semantics and there is no conflict.
- If the programmer specifies a non-temporal store to cacheable memory (for example, Write-Back (WB) or Write-Through (WT) memory types), two cases may result:
  - **Case 1:** If the data is present in the cache hierarchy, the instruction will ensure consistency.
    - **A particular processor may choose different ways to implement this. The following approaches are probable:**
      - Updating data in-place in the cache hierarchy while preserving the memory type semantics assigned to that region or
      - Evicting the data from the caches and writing the new non-temporal data to memory (with WC semantics).
    - **The approaches (separate or combined) can be different for different processors.**
      - If the streaming store hits a line that is present in the first-level cache, the store data is combined in place within the first-level cache.
      - If the streaming store hits a line present in the second-level, the line and stored data is flushed from the second-level to system memory.
  - **Case 2:** If the data is not present in the cache hierarchy and the destination region is mapped as WB or WT
    - **The transaction will be weakly ordered and is subject to all** WC **memory semantics.**
    - **This non-temporal store will not write-allocate.**
    - **Different implementations may choose to collapse and combine such stores.**

9.4.1.4 Write-Combining

Generally, WC semantics require software to ensure coherence with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and a fencing operation must be performed for producer-consumer usage models (see Section 9.4.5). Fencing ensures that all system agents have global visibility of

the stored data. For instance, failure to fence may result in a written cache line staying within a processor, and the line would not be visible to other agents.

For processors which implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. Otherwise, if mapped as WB or WT, there is a potential for speculative processor reads to bring the data into the caches. In such a case, non-temporal stores would then update in place and data would not be flushed from the processor by a subsequent fencing operation.

The memory type visible on the bus in the presence of memory type aliasing is implementation-specific. As one example, the memory type written to the bus may reflect the memory type for the first store to the line, as seen in program order. Other alternatives are possible. This behavior should be considered reserved and dependence on the behavior of any particular implementation risks future incompatibility.

9.4.2 STREAMING STORE USAGE MODELS

The two primary usage domains for streaming store are coherent requests and non-coherent requests.

9.4.2.1 Coherent Requests

Coherent requests are normal loads and stores to system memory, which may also hit cache lines present in another processor in a multiprocessor environment. With coherent requests, a streaming store can be used in the same way as a regular store that has been mapped with a WC memory type (PAT or MTRR). An SFENCE instruction must be used within a producer-consumer usage model to ensure coherency and visibility of data between processors.

Within a single-processor system, the CPU can also re-read the same memory location and be assured of coherence (that is, a single, consistent view of this memory location). The same is true for a multiprocessor (MP) system, assuming an accepted MP software producer-consumer synchronization policy is employed.

9.4.2.2 Non-coherent requests

Non-coherent requests arise from an I/O device, such as an AGP graphics card, that reads or writes system memory using non-coherent requests, which are not reflected on the processor bus and thus will not query the processor's caches. An SFENCE instruction must be used within a producer-consumer usage model to ensure coherency and visibility of data between processors. In this case, if the processor is writing data to the I/O device, a streaming store can be used with a processor with any behavior of Case 1 (Section 9.4.1.3) only if the region has also been mapped with a WC memory type (PAT, MTRR).

NOTE

Failure to map the region as WC may allow the line to be speculatively read into the processor caches (via the wrong path of a mispredicted branch).

In case the region is not mapped as WC, the streaming might update in-place in the cache and a subsequent SFENCE would not result in the data being written to system memory. Explicitly mapping the region as WC in this case ensures that any data read from this region will not be placed in the processor's caches. A read of this memory location by a non-coherent I/O device would return incorrect/out-of-date results.

For a processor which solely implements Case 2 (Section 9.4.1.3), a streaming store can be used in this non-coherent domain without requiring the memory region to also be mapped as WB, since any cached data will be flushed to memory by the streaming store.

9.4.3 STREAMING STORE INSTRUCTION DESCRIPTIONS

MOVNTQ/MOVNTDQ (non-temporal store of packed integer in an MMX technology or Streaming SIMD Extensions register) store data from a register to memory. They are implicitly weakly-ordered, do no write-allocate, and so minimize cache pollution.

MOVNTPS (non-temporal store of packed single precision floating-point) is similar to MOVNTQ. It stores data from a Streaming SIMD Extensions register to memory in 16-byte granularity. Unlike MOVNTQ, the memory address must be aligned to a 16-byte boundary or a general protection exception will occur. The instruction is implicitly weakly-ordered, does not write-allocate, and thus minimizes cache pollution.

MASKMOVQ/MASKMOVDQU (non-temporal byte mask store of packed integer in an MMX technology or Streaming SIMD Extensions register) store data from a register to the location specified by the EDI register. The most significant bit in each byte of the second mask register is used to selectively write the data of the first register on a per-byte basis. The instructions are implicitly weakly-ordered (that is, successive stores may not write memory in original program-order), do not write-allocate, and thus minimize cache pollution.

9.4.4 THE STREAMING LOAD INSTRUCTION

SSE4.1 introduces the MOVNTDQA instruction. MOVNTDQA loads 16 bytes from memory using a non-temporal hint if the memory source is WC type. For WC memory type, the non-temporal hint may be implemented by loading into a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Subsequent MOVNTDQA reads to unread portions of the buffered WC data will cause 16 bytes of data transferred from the temporary internal buffer to an XMM register if data is available.

If used appropriately, MOVNTDQA can help software achieve significantly higher throughput when loading data in WC memory region into the processor than other means.

9.4.5 FENCE INSTRUCTIONS

The following fence instructions are available: **SFENCE**, **lFENCE**, and **MFENCE**.

9.4.5.1 SFENCE Instruction

The **SFENCE** (STORE FENCE) instruction makes it possible for every STORE instruction that precedes an SFENCE in program order to be globally visible before any STORE that follows the SFENCE. SFENCE provides an efficient way of ensuring ordering between routines that produce weakly-ordered results.

The use of weakly-ordered memory types can be important under certain data sharing relationships (such as a producer-consumer relationship). Using weakly-ordered memory can make assembling the data more efficient, but care must be taken to ensure that the consumer obtains the data that the producer intended to see.

Some common usage models may be affected by weakly-ordered stores. Examples are:

- Library functions, which use weakly-ordered memory to write results.
- Compiler-generated code, which also benefits from writing weakly-ordered results.
- Hand-crafted code.

The degree to which a consumer of data knows that the data is weakly-ordered can vary for different cases. As a result, SFENCE should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume this data.

9.4.5.2 LFENCE Instruction

The **LFENCE** (LOAD FENCE) instruction makes it possible for every LOAD instruction that precedes the LFENCE instruction in program order to be globally visible before any LOAD instruction that follows the LFENCE.

The LFENCE instruction provides a means of segregating LOAD instructions from other LOADs.

9.4.5.3 MFENCE Instruction

The **MFENCE** (MEMORY FENCE) instruction makes it possible for every LOAD/STORE instruction preceding MFENCE in program order to be globally visible before any LOAD/STORE following MFENCE. MFENCE provides a means of segregating certain memory instructions from other memory references.

The use of a LFENCE and SFENCE is not equivalent to the use of a MFENCE since the load and store fences are not ordered with respect to each other. In other words, the load fence can be executed before prior stores and the store fence can be executed before prior loads.

MFENCE should be used whenever the cache line flush instruction (CLFLUSH) is used to ensure that speculative memory references generated by the processor do not interfere with the flush. See Section 9.4.6

9.4.6 CLFLUSH INSTRUCTION

The **CLFLUSH** instruction invalidates the cache line associated with the linear address that contain the byte address of the memory location, from all levels of the processor cache hierarchy (data and instruction). This invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, a line is inconsistent with memory (dirty), it is written to memory before invalidation. Other characteristics include:

- The data size affected is the cache coherency size, which is enumerated by the CPUID instruction. It is typically 64 bytes.
- The memory attribute of the page containing the affected line has no effect on the behavior of this instruction.
- The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load.

Executions of the CLFLUSH instruction are ordered with respect to each other, writes locked, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line[1]. They are not ordered with respect to executions of CLFLUSHOPT to different cache lines. For updated memory order details of CLFLUSH and other memory traffic, please refer to the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A sections:

- The CFLUSH reference pages in Chapter 3, "Protected-Mode Memory Management".
- The Memory Ordering pages in Chapter 9, "Multiple-Processor Management".

As an example, consider a video usage model where a video capture device is using non-coherent accesses to write a capture stream directly to system memory. Since these non-coherent writes are not broadcast on the processor bus, they will not flush copies of the same locations that reside in the processor caches. As a result, before the processor re-reads the capture buffer, it should use CLFLUSH to ensure that stale, cached copies of the capture buffer are flushed from the processor caches.

1. Memory order recommendation of CLFLUSH in previous manuals had required software to add MFENCE after CLFLUSH. MFENCE is not required following CLFLUSH as all processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

Example 9-1 provides pseudo-code for CLFLUSH usage.

Example 9-1. Pseudo-code Using CLFLUSH

```
while (!buffer_ready} {}
sfence
    for(i=0;i<num_cachelines;i+=cacheline_size) {
    clflush (char *)((unsigned int)buffer + i)
}
    prefnta buffer[0];
    VAR = buffer[0];
```

The throughput characteristics of using CLFLUSH to flush cache lines can vary significantly depending on several factors. In general using CLFLUSH back-to-back to flush a large number of cache lines will experience larger cost per cache line than flushing a moderately-sized buffer (e.g. less than 4KB); the reduction of CLFLUSH throughput can be an order of magnitude. Flushing cache lines in modified state are more costly than flushing cache lines in non-modified states.

9.4.7 CLFLUSHOPT INSTRUCTION

Similar to CLFLUSH, **CLFLUSHOPT** invalidates the cache line associated with the linear address that contain the byte address of the memory location, in all levels of the processor cache hierarchy (data and instruction).

Executions of the CLFLUSHOPT instruction are ordered with respect to locked read-modify-write instructions, fence instructions, and writes to the cache line being invalidated. (They are also ordered with respect to executions of CLFLUSH and CLFLUSHOPT to the same cache line.) They are not ordered with respect to writes to cache lines other than the one being invalidated. (They are also not ordered with respect to executions of CLFLUSH and CLFLUSHOPT to different cache lines.) Software can insert an SFENCE instruction between CFLUSHOPT and a store to another cache line with which the CLFLUSHOPT should be ordered.

In general, CLFLUSHOPT throughput is higher than that of CLFLUSH, because CLFLUSHOPT orders itself with respect to a smaller set of memory traffic as described above and in Section 9.4.6. The throughput of CLFLUSHOPT will also vary. When using CLFLUSHOPT, flushing modified cache lines will experience a higher cost than flushing cache lines in non-modified states. CLFLUSHOPT will provide a performance benefit over CLFLUSH for cache lines in any coherence states. CLFLUSHOPT is more suitable to flush large buffers (e.g. greater than many KBytes), compared to CLFLUSH. In single-threaded applications, flushing buffers using CLFLUSHOPT may be up to 9X better than using CLFLUSH with Skylake microarchitecture.

Figure 9-1 shows the comparison of the performance characteristics of executing CLFLUSHOPT versus CLFLUSH for buffers of various sizes.

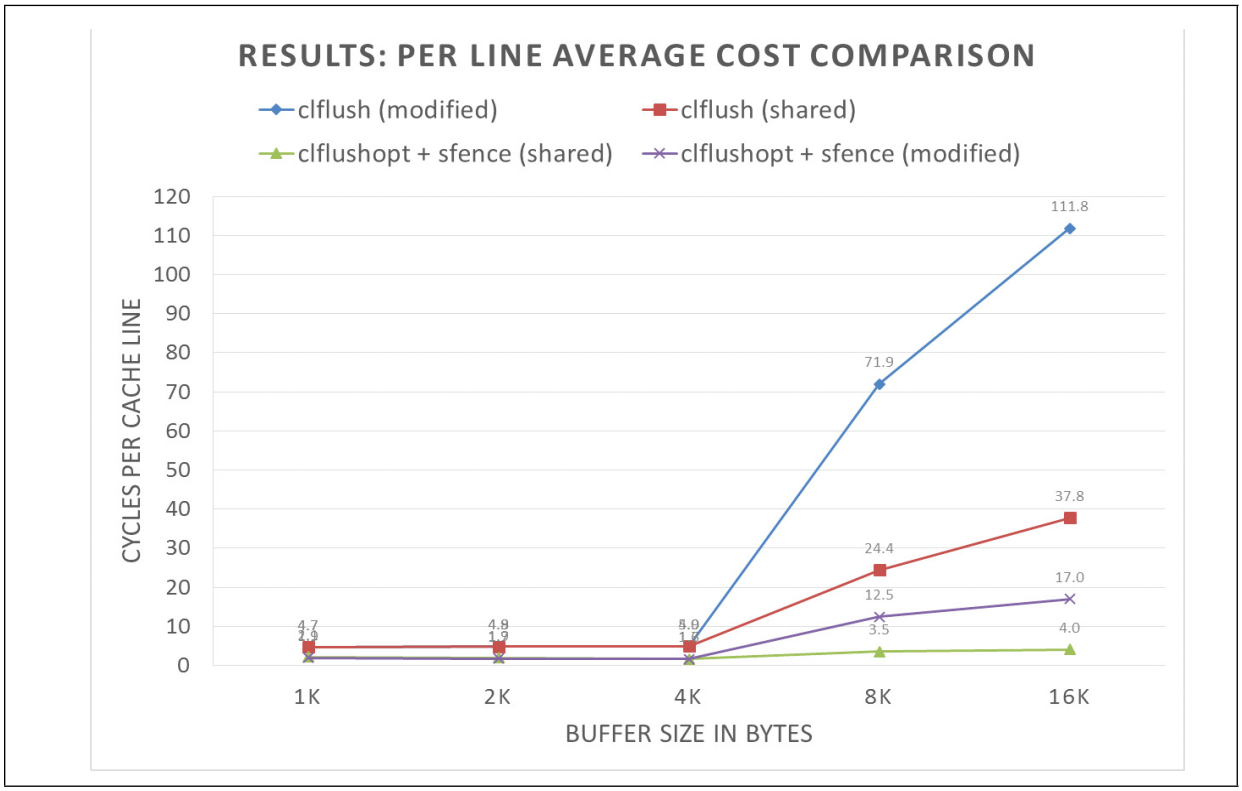


Figure 9-1. CLFLUSHOPT versus CLFLUSH In SkyLake Microarchitecture

User/Source Coding Rule 13. *If CLFLUSHOPT is available, use CLFLUSHOPT over CLFLUSH and use SFENCE to guard CLFLUSHOPT to ensure write order is globally observed. If CLUSHOPT is unavailable, consider flushing large buffers with CLFLUSH in smaller chunks of less than 4KB.*

Example 9-2 gives equivalent assembly sequences of flushing cache lines using CLFLUSH or CLFLUSHOPT. The corresponding sequence in C are:

CLFLUSH:

For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflush(&pBufferToFlush[i]);

CLFLUSHOPT:

_mm_sfence();

For (i = 0; i < iSizeOfBufferToFlush; i += CACHE_LINE_SIZE) _mm_clflushopt(&pBufferToFlush[i]);

_mm_sfence();

Example 9-2. Flushing Cache Lines Using CLFLUSH or CLFLUSHOPT

CLFLUSH no longer requires mfence	CLFLUSHOPT w/ SFENCE

xor rcx, rcx
mov r9, pBufferToFlush
mov rsi, iSizeOfBufferToFlush
;; mfence - obsolete
loop:
clflush [r9+rcx]
add rcx, 0x40
cmp rcx, rsi
jl loop
;; mfence - obsolete
	xor rcx, rcx
	mov r9, pBufferToFlush
	mov rsi, iSizeOfBufferToFlush
	sfence
	loop:
	clflushopt [r9+rcx]
	add rcx, 0x40
	cmp rcx, rsi
	jl loop
	sfence

* If imposing memory ordering rules is important for the application then executing CLFLUSHOPT instructions should be guarded with SFENCE instructions to guarantee order of memory writes. As per the example above, such solution still performs better than using the CLFLUSH instruction, and its performance is identical to CLFLUSHOPT from 2048 byte buffers and bigger.	

9.5 MEMORY OPTIMIZATION USING PREFETCH

Recent generations of Intel processors have two mechanisms for data prefetch: software-controlled prefetch and an automatic hardware prefetch.

9.5.1 SOFTWARE-CONTROLLED PREFETCH

The software-controlled prefetch is enabled using the four PREFETCH instructions introduced with Streaming SIMD Extensions instructions. These instructions are hints to bring a cache line of data in to various levels and modes in the cache hierarchy. The software-controlled prefetch is not intended for prefetching code. Using it can incur significant penalties on a multiprocessor system when code is shared.

Software prefetching has the following characteristics:

- Can handle irregular access patterns which do not trigger the hardware prefetcher.
- Can use less bus bandwidth than hardware prefetching; see below.
- Software prefetches must be added to new code, and do not benefit existing applications.

9.5.2 HARDWARE PREFETCH

Automatic hardware prefetch can bring cache lines into the unified last-level cache based on prior data misses. It will attempt to prefetch two cache lines ahead of the prefetch stream. Characteristics of the hardware prefetcher are:

- It requires some regularity in the data access patterns.
  - — If a data access pattern has constant stride, hardware prefetching is effective if the access stride is less than half of the trigger distance of hardware prefetcher.
  - — If the access stride is not constant, the automatic hardware prefetcher can mask memory latency if the strides of two successive cache misses are less than the trigger threshold distance (small-stride memory traffic).

  - The automatic hardware prefetcher is most effective if the strides of two successive cache misses remain less than the trigger threshold distance and close to 64 bytes.
- There is a start-up penalty before the prefetcher triggers and there may be fetches an array finishes. For short arrays, overhead can reduce effectiveness.
  - The hardware prefetcher requires a couple misses before it starts operating.
  - Hardware prefetching generates a request for data beyond the end of an array, which is not be utilized. This behavior wastes bus bandwidth. In addition this behavior results in a start-up penalty when fetching the beginning of the next array. Software prefetching may recognize and handle these cases.
- It will not prefetch across a 4-KByte page boundary. A program has to initiate demand loads for the new page before the hardware prefetcher starts prefetching from the new page.
- The hardware prefetcher may consume extra system bandwidth if the application's memory traffic has significant portions with strides of cache misses greater than the trigger distance threshold of hardware prefetch (large-stride memory traffic).
- The effectiveness with existing applications depends on the proportions of small-stride versus large-stride accesses in the application's memory traffic. An application with a preponderance of small-stride memory traffic with good temporal locality will benefit greatly from the automatic hardware prefetcher.
- In some situations, memory traffic consisting of a preponderance of large-stride cache misses can be transformed by re-arrangement of data access sequences to alter the concentration of small-stride cache misses at the expense of large-stride cache misses to take advantage of the automatic hardware prefetcher.

9.5.3 EXAMPLE OF EFFECTIVE LATENCY REDUCTION WITH HARDWARE PREFETCH

Consider the situation that an array is populated with data corresponding to a constant-access-stride, circular pointer chasing sequence (see Example 9-3). The potential of employing the automatic hardware prefetching mechanism to reduce the effective latency of fetching a cache line from memory can be illustrated by varying the access stride between 64 bytes and the trigger threshold distance of hardware prefetch when populating the array for circular pointer chasing.

Example 9-3. Populating an Array for Circular Pointer Chasing with Constant Stride

```
register char ** p;
char  *next;       // Populating pArray for circular pointer
                   //  chasing with constant access stride
                   // p = (char **) *p; loads a value pointing to next load
    p = (char **)&pArray;

    for ( i = 0; i < aperture; i += stride) {
          p = (char **)&pArray[i];
          if (i + stride >= g_array_aperture) {
                next = &pArray[0 ];
                }
    else {
                next = &pArray[i + stride];
                }
    *p = next; // populate the address of the next node
    }
```

The effective latency reduction for several microarchitecture implementations is shown in Figure 9-2. For a constant-stride access pattern, the benefit of the automatic hardware prefetcher begins at half the trigger threshold distance and reaches maximum benefit when the cache-miss stride is 64 bytes.

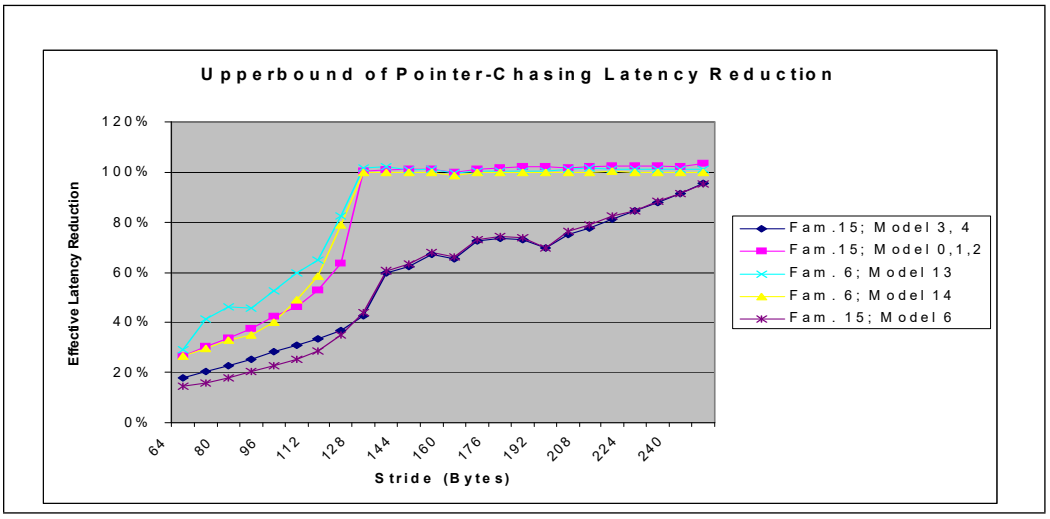


Figure 9-2. Effective Latency Reduction as a Function of Access Stride

9.5.4 EXAMPLE OF LATENCY HIDING WITH S/W PREFETCH INSTRUCTION

Achieving the highest level of memory optimization using PREFETCH instructions requires an understanding of the architecture of a given machine. This section translates the key architectural implications into several simple guidelines for programmers to use.

Figure 9-3 and Figure 9-4 show two scenarios of a simplified 3D geometry pipeline as an example. A 3D-geometry pipeline typically fetches one vertex record at a time and then performs transformation and lighting functions on it. Both figures show two separate pipelines, an execution pipeline, and a memory pipeline (front-side bus).

Since the processor completely decouples the functionality of execution and memory access, the two pipelines can function concurrently. Figure 9-3 shows “bubbles” in both the execution and memory pipelines. When loads are issued for accessing vertex data, the execution units sit idle and wait until data is returned. On the other hand, the memory bus sits idle while the execution units are processing vertices. This scenario severely decreases the advantage of having a decoupled architecture.

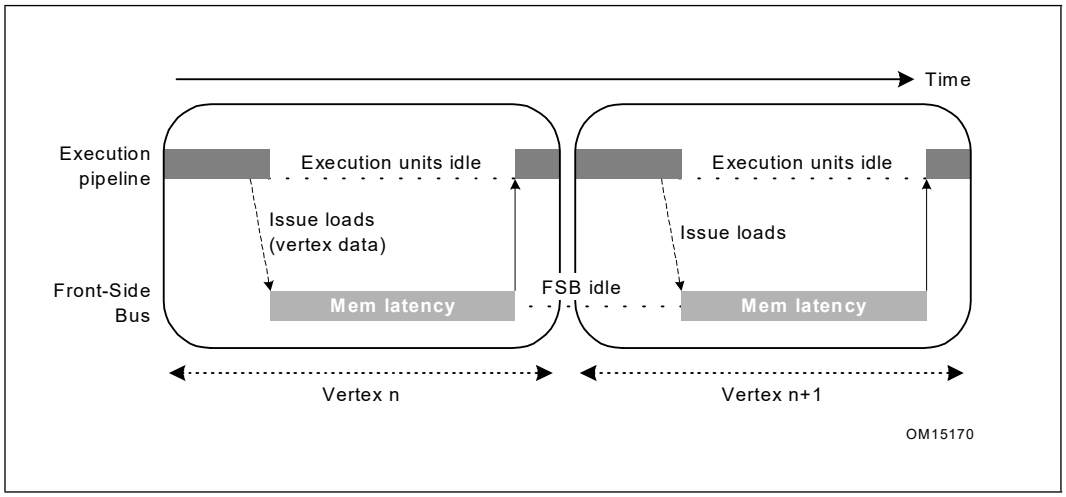


Figure 9-3. Memory Access Latency and Execution Without Prefetch

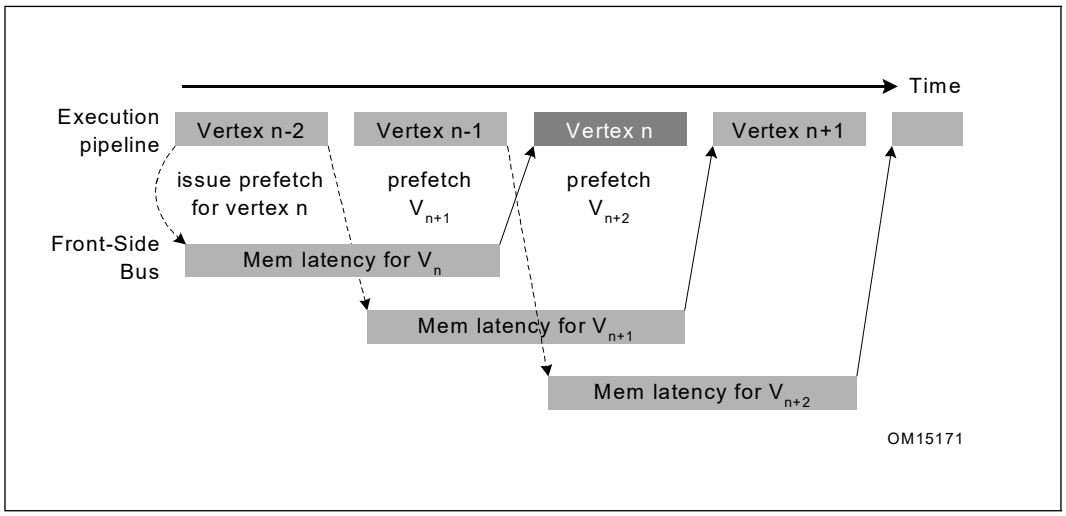


Figure 9-4. Memory Access Latency and Execution With Prefetch

The performance loss caused by poor utilization of resources can be completely eliminated by correctly scheduling the PREFETCH instructions. As shown in Figure 9-4, prefetch instructions are issued two vertex iterations ahead. This assumes that only one vertex gets processed in one iteration and a new data cache line is needed for each iteration. As a result, when iteration n, vertex V_n, is being processed; the requested data is already brought into cache. In the meantime, the front-side bus is transferring the data needed for iteration n+1, vertex V_{n+1}. Because there is no dependence between V_{n+1} data and the execution of V_n, the latency for data access of V_{n+1} can be entirely hidden behind the execution of V_n. Under such circumstances, no “bubbles” are present in the pipelines and thus the best possible performance can be achieved.

Prefetching is useful for inner loops that have heavy computations, or are close to the boundary between being compute-bound and memory-bandwidth-bound. It is probably not very useful for loops which are predominately memory bandwidth-bound.

When data is already located in the first level cache, prefetching can be useless and could even slow down the performance because the extra µops either back up waiting for outstanding memory accesses or may be dropped altogether. This behavior is platform-specific and may change in the future.

9.5.5 SOFTWARE PREFETCHING USAGE CHECKLIST

The following checklist covers issues that need to be addressed and/or resolved to use the software PREFETCH instruction properly:

- Determine software prefetch scheduling distance.
- Use software prefetch concatenation.
- Minimize the number of software prefetches.
- Mix software prefetch with computation instructions.
- Use cache blocking techniques (for example, strip mining).
- Balance single-pass versus multi-pass execution.
- Resolve memory bank conflict issues.
- Resolve cache management issues.

Subsequent sections discuss the above items.

9.5.6 SOFTWARE PREFETCH SCHEDULING DISTANCE

Determining the ideal prefetch placement in the code depends on many architectural parameters, including: the amount of memory to be prefetched, cache lookup latency, system memory latency, and estimate of computation cycle. The ideal distance for prefetching data is processor- and platform-dependent. If the distance is too short, the prefetch will not hide the latency of the fetch behind computation. If the prefetch is too far ahead, prefetched data may be flushed out of the cache by the time it is required.

Since prefetch distance is not a well-defined metric, for this discussion, we define a new term, prefetch scheduling distance (PSD), which is represented by the number of iterations. For large loops, prefetch scheduling distance can be set to 1 (that is, schedule prefetch instructions one iteration ahead). For small loop bodies (that is, loop iterations with little computation), the prefetch scheduling distance must be more than one iteration.

A simplified equation to compute PSD is deduced from the mathematical model.

Example 9-4 illustrates the use of a prefetch within the loop body. The prefetch scheduling distance is set to 3, ESI is effectively the pointer to a line, EDX is the address of the data being referenced and XMM1-XMM4 are the data used in computation. Example 9-5 uses two independent cache lines of data per iteration. The PSD would need to be increased/decreased if more/less than two cache lines are used per iteration.

Example 9-4. Prefetch Scheduling Distance

```
top_loop:
    prefetchnta [edx + esi + 128*3]
    prefetchnta [edx*4 + esi + 128*3]
    . . . . .
    movaps  xmm1, [edx + esi]
    movaps  xmm2, [edx*4 + esi]
    movaps  xmm3, [edx + esi + 16]
    movaps  xmm4, [edx*4 + esi + 16]

    . . . . .
    . . . . .
    add     esi, 128
    cmp     esi, ecx
    jl      top_loop
```

9.5.7 SOFTWARE PREFETCH CONCATENATION

Maximum performance can be achieved when the execution pipeline is at maximum throughput, without incurring any memory latency penalties. This can be achieved by prefetching data to be used in successive iterations in a loop. De-pipelining memory generates bubbles in the execution pipeline.

To explain this performance issue, a 3D geometry pipeline that processes 3D vertices in strip format is used as an example. A strip contains a list of vertices whose predefined vertex order forms contiguous triangles. It can be easily observed that the memory pipe is de-pipelined on the strip boundary due to ineffective prefetch arrangement. The execution pipeline is stalled for the first two iterations for each strip. As a result, the average latency for completing an iteration will be 165 (FIX) clocks.

This memory de-pipelining creates inefficiency in both the memory pipeline and execution pipeline. This de-pipelining effect can be removed by applying a technique called prefetch concatenation. With this technique, the memory access and execution can be fully pipelined and fully utilized.

For nested loops, memory de-pipelining could occur during the interval between the last iteration of an inner loop and the next iteration of its associated outer loop. Without paying special attention to prefetch insertion, loads from the first iteration of an inner loop can miss the cache and stall the execution pipeline waiting for data returned, thus degrading the performance.

In Example 9-5, the cache line containing A[II][0] is not prefetched at all and always misses the cache. This assumes that no array A[][] footprint resides in the cache. The penalty of memory de-pipelining stalls can be amortized across the inner loop iterations. However, it may become very harmful when the inner loop is short. In addition, the last prefetch in the last PSD iterations are wasted and consume machine resources. Prefetch concatenation is introduced here to eliminate the performance issue of memory de-pipelining.

Example 9-5. Using Prefetch Concatenation

```
for (ii = 0; ii < 100; ii++) {
  for (jj = 0; jj < 32; jj+=8) {
       prefetch a[ii][jj+8]
       computation a[ii][jj]
  }
}
```

Prefetch concatenation can bridge the execution pipeline bubbles between the boundary of an inner loop and its associated outer loop. Simply by unrolling the last iteration out of the inner loop and specifying the effective prefetch address for data used in the following iteration, the performance loss of memory de-pipelining can be completely removed. Example 9-6 gives the rewritten code.

Example 9-6. Concatenation and Unrolling the Last Iteration of Inner Loop

```
for (ii = 0; ii < 100; ii++) {
  for (jj = 0; jj < 24; jj+=8) { /* N-1 iterations */
       prefetch a[ii][jj+8]
       computation a[ii][jj]
  }
  prefetch a[ii+1][0]
  computation a[ii][jj]/* Last iteration */
}
```

This code segment for data prefetching is improved and only the first iteration of the outer loop suffers any memory access latency penalty, assuming the computation time is larger than the memory latency. Inserting a prefetch of the first data element needed prior to entering the nested loop computation would eliminate or reduce the start-up penalty for the very first iteration of the outer loop. This uncomplicated high-level code optimization can improve memory performance significantly.

9.5.8 MINIMIZE NUMBER OF SOFTWARE PREFETCHES

Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources, even though they require minimal clock and memory bandwidth.

Excessive prefetching may lead to performance penalties because of issue penalties in the front end of the machine and/or resource contention in the memory sub-system. This effect may be severe in cases where the target loops are small and/or cases where the target loop is issue-bound.

One approach to solve the excessive prefetching issue is to unroll and/or software-pipeline loops to reduce the number of prefetches required. Figure 9-5 presents a code example which implements prefetch and unrolls the loop to remove the redundant prefetch instructions whose prefetch addresses hit the previously issued prefetch instructions. In this particular example, unrolling the original loop once saves six prefetch instructions and nine instructions for conditional jumps in every other iteration.

```
top_loop:
prefetchnta  [edx+esi+32]
prefetchnta  [edx*4+esi+32]
. . . . . .
movaps xmm1,  [edx+esi]
movaps xmm2, [edx*4+esi]
. . . . . .
add esi, 16
cmp esi, ecx
jl top_loop
```

unrolled iteration

```
top_loop:
prefetchnta  [edx+esi+128]
prefetchnta  [edx*4+esi+128]
. . . . . .
movaps xmm1,  [edx+esi]
movaps xmm2, [edx*4+esi]
. . . . . .
movaps xmm1, [edx+esi+16]
movaps xmm2, [edx*4+esi+16]
. . . . . .
movaps xmm1, [edx+esi+96]
movaps xmm2, [edx*4+esi+96]
. . . . . .
. . . . . .
add esi, 128
cmp esi, ecx
jl top_loop
```

Figure 9-5. Prefetch and Loop Unrolling

Figure 9-6 demonstrates the effectiveness of software prefetches in latency hiding.

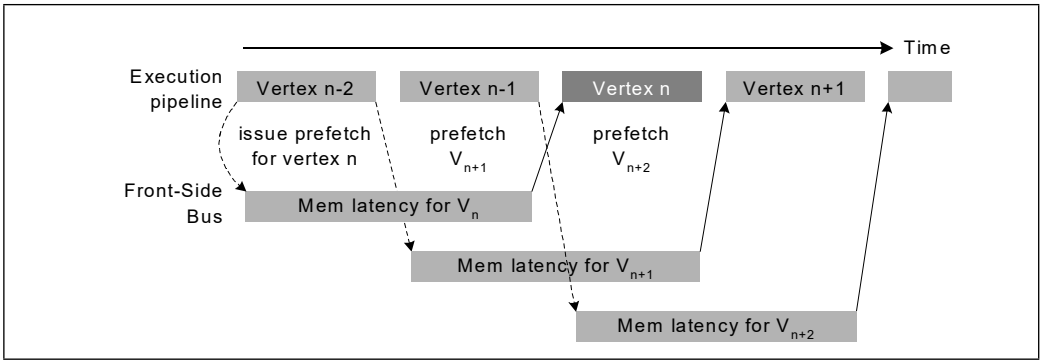


Figure 9-6. Memory Access Latency and Execution With Prefetch

The Xaxis in Figure 9-6 indicates the number of computation clocks per loop (each iteration is independent). The Y axis indicates the execution time measured in clocks per loop. The secondary Y-axis indicates the percentage of bus bandwidth utilization. The tests vary by the following parameters:

- **Number of Load /Store Streams:** Each load and store stream accesses one 128-byte cache line each per iteration.
- **Amount of Computation Per Loop:** This is varied by increasing the number of dependent arithmetic operations executed.

- **Number of the software prefetches per loop** — For example, one every 16 bytes, 32 bytes, 64 bytes, 128 bytes.

As expected, the leftmost portion of each of the graphs in Figure 9-6 shows that when there is not enough computation to overlap the latency of memory access, prefetch does not help and that the execution is essentially memory-bound. The graphs also illustrate that redundant prefetches do not increase performance.

9.5.9 MIX SOFTWARE PREFETCH WITH COMPUTATION INSTRUCTIONS

It may seem convenient to cluster all of PREFETCH instructions at the beginning of a loop body or before a loop, but this can lead to severe performance degradation. to achieve the best possible performance, PREFETCH instructions must be interspersed with other computational instructions in the instruction sequence rather than clustered together. If possible, they should also be placed apart from loads. This improves the instruction level parallelism and reduces the potential instruction resource stalls. In addition, this mixing reduces the pressure on the memory access resources and in turn reduces the possibility of the prefetch retiring without fetching data.

Figure 9-7 illustrates distributing PREFETCH instructions. Rearranging PREFETCH instructions could yield a noticeable speedup for the code which stresses the cache resource.

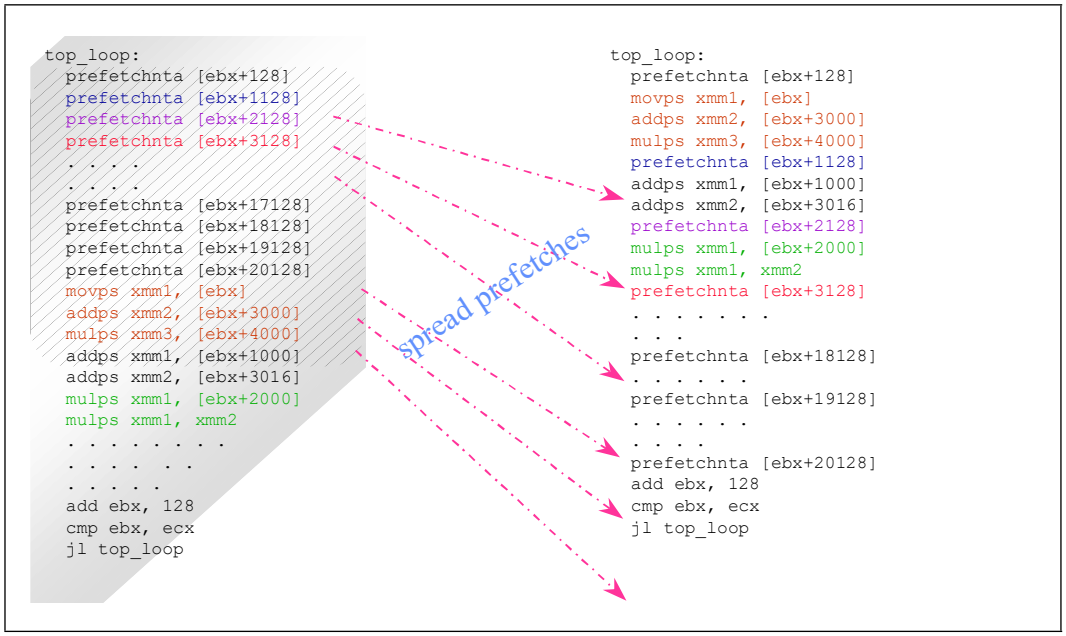


Figure 9-7. Spread Prefetch Instructions

NOTE

To avoid instruction execution stalls due to the over-utilization of the resource, PREFETCH instructions must be interspersed with computational instructions. The spreading of PREFETCH instructions may need to be re-tuned for new processors.

9.5.10 SOFTWARE PREFETCH AND CACHE BLOCKING TECHNIQUES

Cache blocking techniques (such as strip-mining) are used to improve temporal locality and the cache hit rate. Strip-mining is one-dimensional temporal locality optimization for memory. When higher-dimensional arrays are used in programs, loop blocking technique (similar to strip-mining but in two dimensions) can be applied for a better memory performance.

If an application uses a large data set that can be reused across multiple passes of a loop, it will benefit from strip mining. Data sets larger than the cache will be processed in groups small enough to fit into cache. This allows temporal data to reside in the cache longer, reducing bus traffic.

Data set size and temporal locality (data characteristics) fundamentally affect how PREFETCH instructions are applied to strip-mined code. Figure 9-8 shows two simplified scenarios for temporally-adjacent data and temporally-non-adjacent data.

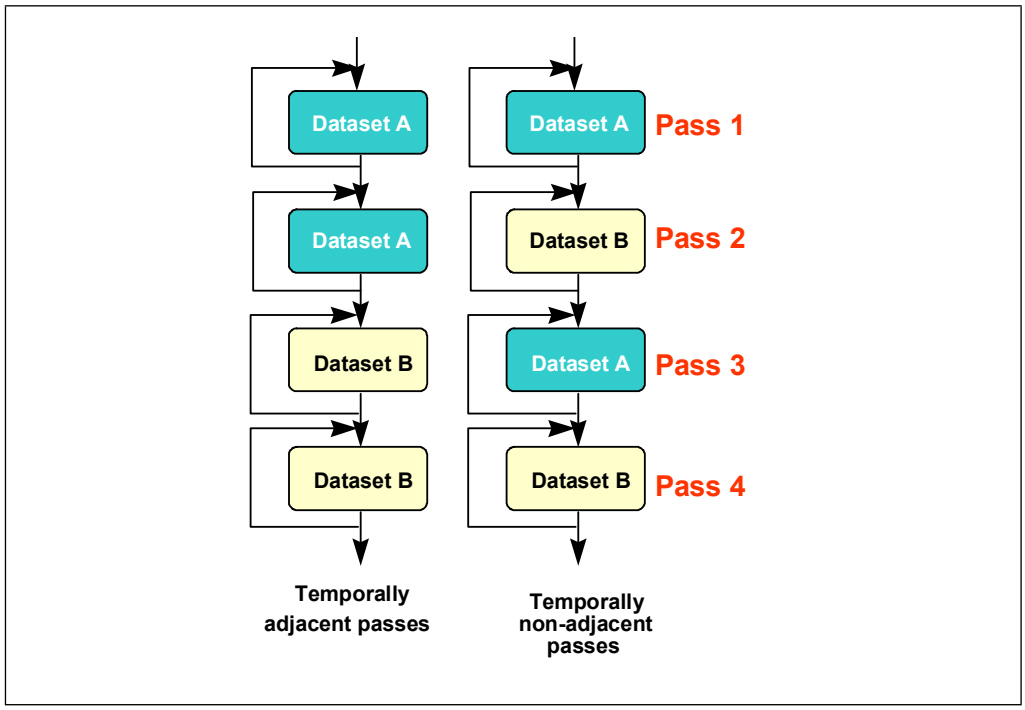


Figure 9-8. Cache Blocking – Temporally Adjacent and Non-adjacent Passes

In the temporally-adjacent scenario, subsequent passes use the same data and find it already in second-level cache. Prefetch issues aside, this is the preferred situation. In the temporally non-adjacent scenario, data used in pass *m* is displaced by pass *(m+1)*, requiring data *re*-fetch into the first level cache and perhaps the second level cache if a later pass reuses the data. If both data sets fit into the second-level cache, load operations in passes 3 and 4 become less expensive.

Figure 9-9 shows how prefetch instructions and strip-mining can be applied to increase performance in both of these scenarios.

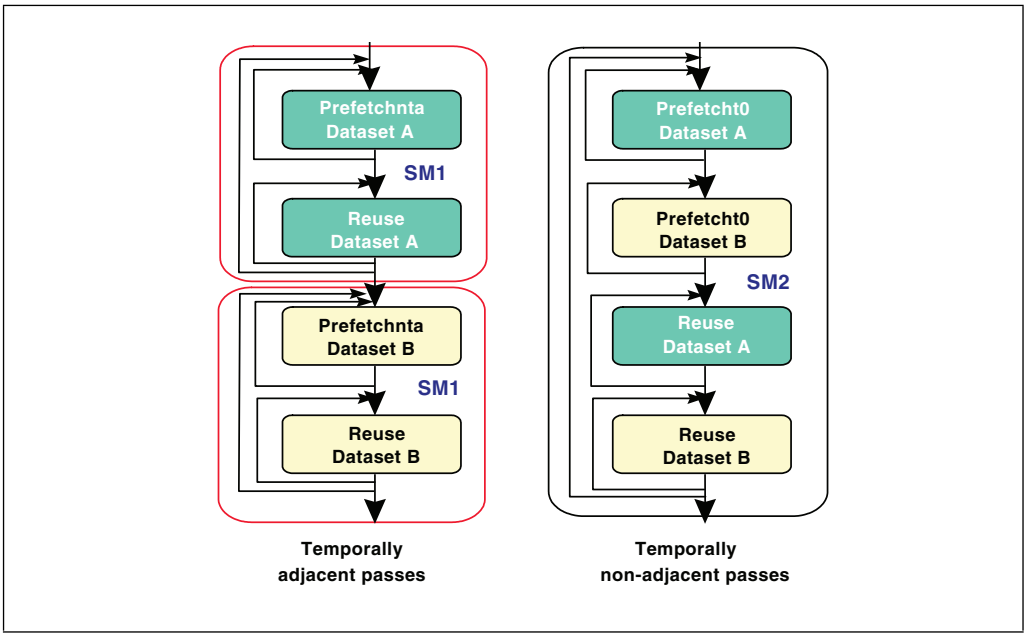


Figure 9-9. Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops

The left scenario shows a graphical implementation of using **PREFETCHNTA** to prefetch data into L1, minimizing second-level cache pollution. Use PREFETCHNTA if the data is only touched once during the entire execution pass to minimize cache pollution in the higher level caches. This provides instant availability, assuming the prefetch was issued far ahead enough, when the read access is issued.

In the scenario to the right (see Figure 9-9), the workload footprint is too large for the L1 cache. Therefore, use PREFETCHT0 to prefetch the data. This amortizes the latency of the memory references in passes 1 and 2, and keeps a copy of the data in second-level cache, which reduces memory traffic and latencies for passes 3 and 4. To further reduce the latency, it might be worth considering extra PREFETCHNTA instructions prior to the memory references in passes 3 and 4.

In Example 9-7, consider the data access patterns of a 3D geometry engine first without strip-mining and then incorporating strip-mining.

Without strip-mining, all the x,y,z coordinates for the four vertices must be re-fetched from memory in the second pass, that is, the lighting loop. This causes under-utilization of cache lines fetched during transformation loop as well as bandwidth wasted in the lighting loop.

Example 9-7. Data Access of a 3D Geometry Engine without Strip-mining

```
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data          // v =[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
TRANSFORMATION code                   // use only x,y,z,tu,tv of a vertex
    nvtx+=4
```

Example 9-7. Data Access of a 3D Geometry Engine without Strip-mining (Contd.)

```
}
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data          // v =[x,y,z,nx,ny,nz,tu,tv]
                                      // x,y,z fetched again
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    compute the light vectors         // use only x,y,z
    LOCAL LIGHTING code               // use only nx,ny,nz
    nvtx+=4
}
```

Now consider the code in Example 9-8 where strip-mining has been incorporated into the loops.

Example 9-8. Data Access of a 3D Geometry Engine with Strip-mining

```
while (nstrip < NUM_STRIP) {
/* Strip-mine the loop to fit data into one way of the second-level
  cache */
 while (nvtx < MAX_NUM_VTX_PER_STRIP) {
    prefetchnta vertex_i data                 // v=[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertex_i+1 data
    prefetchnta vertex_i+2 data
    prefetchnta vertex_i+3 data
    TRANSFORMATION code
         nvtx+=4
}
while (nvtx < MAX_NUM_VTX_PER_STRIP) {
   /* x y z coordinates are in the second-level cache, no prefetch is
      required */
compute the light vectors
POINT LIGHTING code
   nvtx+=4
 }
}
```

With strip-mining, all vertex data can be kept in the cache (for example, one way of second-level cache) during the strip-mined transformation loop and reused in the lighting loop. Keeping data in the cache reduces both bus traffic and the number of prefetches used.

Table 9-4 summarizes the steps of the basic usage model that incorporates only software prefetch with strip-mining. The steps are:

- Do strip-mining: partition loops so that the dataset fits into second-level cache.
- Use PREFETCHNTA if the data is only used once or the dataset fits into 32 KBytes (one way of second-level cache). Use PREFETCHT0 if the dataset exceeds 32 KBytes.

The above steps are platform-specific and provide an implementation example. The variables NUM_STRIP and MAX_NUM_VX_PER_STRIP can be heuristically determined for peak performance for specific application on a specific platform.

Table 9-4. Software Prefetching Considerations into Strip-mining Code

Read-Once Array References	Read-Multiple-Times Array References Adjacent Passes	Read-Multiple-Times Array References Non-Adjacent Passes
Prefetchnta	Prefetch0, SM1	Prefetch0, SM1 (2nd Level Pollution)
• Evict one way • Minimize pollution	• Pay memory access cost for the first pass of each array;. • Amortize the first pass with subsequent passes.	• Pay memory access cost for the first pass of every strip. • Amortize the first pass with subsequent passes.

9.5.11 HARDWARE PREFETCHING AND CACHE BLOCKING TECHNIQUES

Tuning data access patterns for the automatic hardware prefetch mechanism can minimize the memory access costs of the first-pass of the read-multiple-times and some of the read-once memory references. An example of the situations of read-once memory references can be illustrated with a matrix or image transpose, reading from a column-first orientation and writing to a row-first orientation, or vice versa.

Example 9-9 shows a nested loop of data movement that represents a typical matrix/image transpose problem. If the dimension of the array are large, not only the footprint of the dataset will exceed the last level cache but cache misses will occur at large strides. If the dimensions happen to be powers of 2, aliasing condition due to finite number of way-associativity (see Section 3.6.7) will exacerbate the likelihood of cache evictions.

Example 9-9. Using HW Prefetch to Improve Read-Once Memory Traffic

```
a)  Un-optimized image transpose
// dest and src represent two-dimensional arrays
for( i = 0;i < NUMCOLS; i ++) {
    // inner loop reads single column
    for( j = 0; j < NUMROWS ; j ++) {
        // Each read reference causes large-stride cache miss
        dest[i*NUMROWS +j] = src[j*NUMROWS + i];
    }
}
b)
// tilewidth = L2SizeInBytes/2/TileHeight/Sizeof(element)
for( i = 0; i < NUMCOLS; i += tilewidth) {
 for( j = 0; j < NUMROWS ; j ++) {
    // access multiple elements in the same row in the inner loop
    // access pattern friendly to hw prefetch and improves hit rate
    for( k = 0; k < tilewidth; k ++)
    dest[j+ (i+k)* NUMROWS] = src[i+k+ j* NUMROWS];
  }
}
```

Example 9-9 (b) shows applying the techniques of tiling with optimal selection of tile size and tile width to take advantage of hardware prefetch. With tiling, one can choose the size of two tiles to fit in the last level cache. Maximizing the width of each tile for memory read references enables the hardware prefetcher to initiate bus requests to read some cache lines before the code actually reference the linear addresses.

9.5.12 SINGLE-PASS VERSUS MULTI-PASS EXECUTION

An algorithm can use single- or multi-pass execution defined as follows:

- Single-pass, or unlayered execution passes a single data element through an entire computation pipeline.
- Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements, before passing the batch on to the next stage.

A characteristic feature of both single-pass and multi-pass execution is that a specific trade-off exists depending on an algorithm's implementation and use of a single-pass or multiple-pass execution. See Figure 9-10.

Multi-pass execution is often easier to use when implementing a general purpose API, where the choice of code paths that can be taken depends on the specific combination of features selected by the application (for example, for 3D graphics, this might include the type of vertex primitives used and the number and type of light sources).

With such a broad range of permutations possible, a single-pass approach would be complicated, in terms of code size and validation. In such cases, each possible permutation would require a separate code sequence. For example, an object with features A, B, C, D can have a subset of features enabled, say, A, B, D. This stage would use one code path; another combination of enabled features would have a different code path. It makes more sense to perform each pipeline stage as a separate pass, with conditional clauses to select different features that are implemented within each stage. By using strip-mining, the number of vertices processed by each stage (for example, the batch size) can be selected to ensure that the batch stays within the processor caches through all passes. An intermediate cached buffer is used to pass the batch of vertices from one stage or pass to the next one.

Single-pass execution can be better suited to applications which limit the number of features that may be used at a given time. A single-pass approach can reduce the amount of data copying that can occur with a multi-pass engine. See Figure 9-10.

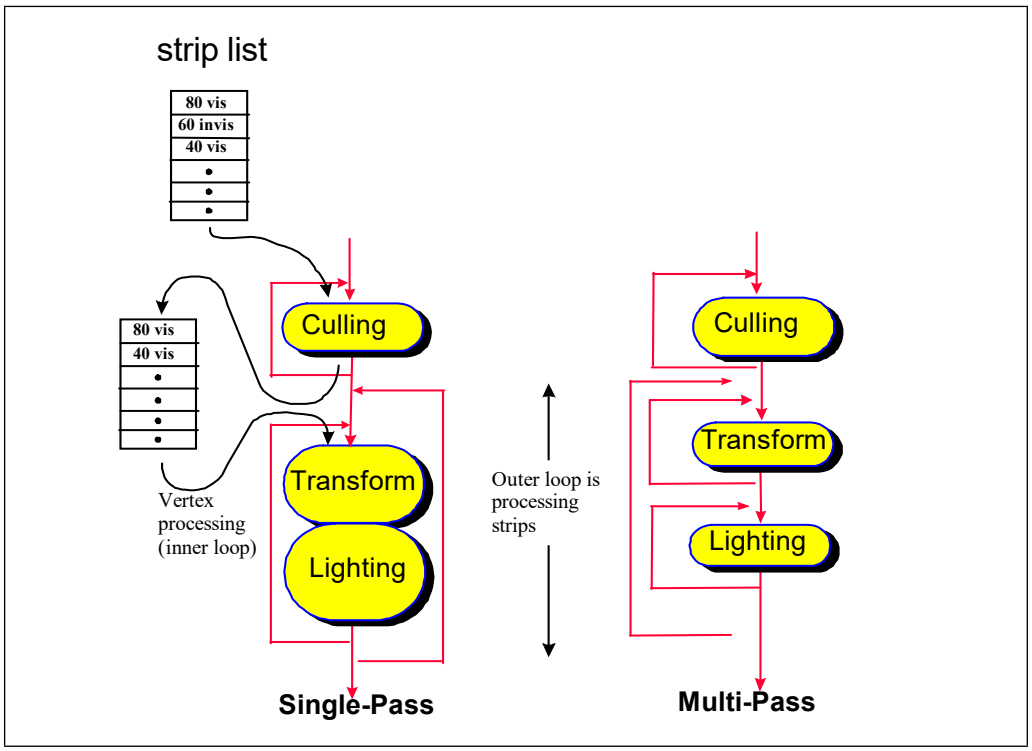


Figure 9-10. Single-Pass vs. Multi-Pass 3D Geometry Engines

The choice of single-pass or multi-pass can have several performance implications. For instance, in a multi-pass pipeline, stages that are limited by bandwidth (either input or output) will reflect more of this performance limitation in overall execution time. In contrast, for a single-pass approach, bandwidth-limitations can be distributed/amortized across other computation-intensive stages. Also, the choice of which prefetch hints to use are also impacted by whether a single-pass or multi-pass approach is used.

9.6 MEMORY OPTIMIZATION USING NON-TEMPORAL STORES

Non-temporal stores can also be used to manage data retention in the cache. Uses for non-temporal stores include:

- To combine many writes without disturbing the cache hierarchy.
- To manage which data structures remain in the cache and which are transient.

Detailed implementations of these usage models are covered in the following sections.

9.6.1 NON-TEMPORAL STORES AND SOFTWARE WRITE-COMBINING

Use non-temporal stores in the cases when the data to be stored is:

- Write-once (non-temporal).
- Too large and thus cause cache thrashing.

Non-temporal stores do not invoke a cache line allocation, which means they are not write-allocate. As a result, caches are not polluted and no dirty writeback is generated to compete with useful data bandwidth. Without using non-temporal stores, bus bandwidth will suffer when caches start to be thrashed because of dirty writebacks.

In Streaming SIMD Extensions implementation, when non-temporal stores are written into writeback or write-combining memory regions, these stores are weakly-ordered and will be combined internally inside the processor's write-combining buffer and be written out to memory as a line burst transaction. To achieve the best possible performance, it is recommended to align data along the cache line boundary and write them consecutively in a cache line size while using non-temporal stores. If the consecutive writes are prohibitive due to programming constraints, then software write-combining (SWWC) buffers can be used to enable line burst transaction.

You can declare small SWWC buffers (a cache line for each buffer) in your application to enable explicit write-combining operations. Instead of writing to non-temporal memory space immediately, the program writes data into SWWC buffers and combines them inside these buffers. The program only writes a SWWC buffer out using non-temporal stores when the buffer is filled up, that is, a cache line. Although the SWWC method requires explicit instructions for performing temporary writes and reads, this ensures that the transaction on the front-side bus causes line transaction rather than several partial transactions. Application performance gains considerably from implementing this technique. These SWWC buffers can be maintained in the second-level and re-used throughout the program.

9.6.2 CACHE MANAGEMENT

Streaming instructions (PREFETCH and STORE) can be used to manage data and minimize disturbance of temporal data held within the processor's caches.

In addition, the processor takes advantage of Intel C ++ Compiler support for C ++ language-level features for the Streaming SIMD Extensions. Streaming SIMD Extensions and MMX technology instructions provide intrinsics that allow you to optimize cache utilization. Examples of such Intel compiler intrinsics are _MM_PREFETCH, _MM_STREAM, _MM_LOAD, _MM_SFENCE. .

The following examples of using prefetching instructions in the operation of video encoder and decoder as well as in simple 8-byte memory copy, illustrate performance gain from using the prefetching instructions for efficient cache management.

9.6.2.1 Video Encoder

In a video encoder, some of the data used during the encoding process is kept in the processor's second-level cache. This is done to minimize the number of reference streams that must be re-read from system memory. To ensure that other writes do not disturb the data in the second-level cache, streaming stores (MOVNTQ) are used to write around all processor caches.

The prefetching cache management implemented for the video encoder reduces the memory traffic. The second-level cache pollution reduction is ensured by preventing single-use video frame data from entering the second-level cache. Using a non-temporal PREFETCH (PREFETCHNTA) instruction brings data into the first-level cache, thus reducing pollution of the second-level cache.

If the data brought directly to second-level cache is not re-used, then there is a performance gain from the non-temporal prefetch over a temporal prefetch. The encoder uses non-temporal prefetches to avoid pollution of the second-level cache, increasing the number of second-level cache hits and decreasing the number of polluting write-backs to memory. The performance gain results from the more efficient use of the second-level cache, not only from the prefetch itself.

9.6.2.2 Video Decoder

In the video decoder example, completed frame data is written to local memory of the graphics card, which is mapped to WC (Write-combining) memory type. A copy of reference data is stored to the WB memory at a later time

by the processor to generate future data. The assumption is that the size of the reference data is too large to fit in the processor's caches. A streaming store is used to write the data around the cache, to avoid displaying other temporal data held in the caches. Later, the processor re-reads the data using PREFETCHNTA, which ensures maximum bandwidth, yet minimizes disturbance of other cached temporal data by using the non-temporal (NTA) version of prefetch.

9.6.2.3 Conclusions from Video Encoder and Decoder Implementation

These two examples indicate that by using an appropriate combination of non-temporal prefetches and non-temporal stores, an application can be designed to lessen the overhead of memory transactions by preventing second-level cache pollution, keeping useful data in the second-level cache and reducing costly write-back transactions. Even if an application does not gain performance significantly from having data ready from prefetches, it can improve from more efficient use of the second-level cache and memory. Such design reduces the encoder's demand for such critical resource as the memory bus. This makes the system more balanced, resulting in higher performance.

9.6.2.4 Optimizing Memory Copy Routines

Creating memory copy routines for large amounts of data is a common task in software optimization. Example 9-10 presents a basic algorithm for a simple memory copy.

Example 9-10. Basic Algorithm of a Simple Memory Copy

```
#define N 512000
double a[N], b[N];
for (i = 0; i < N; i++) {
        b[i] = a[i];
}
```

This task can be optimized using various coding techniques. One technique uses software prefetch and streaming store instructions. It is discussed in the following paragraph and a code example shown in Example 9-11.

The memory copy algorithm can be optimized using the Streaming SIMD Extensions with these considerations:

- Alignment of data.
- Proper layout of pages in memory.
- Cache size.
- Interaction of the transaction lookaside buffer (TLB) with memory accesses.
- Combining prefetch and streaming-store instructions.

Example 9-11. A Memory Copy Routine Using Software Prefetch

```
#define PAGESIZE 4096;
#define NUMPERPAGE 512              // # of elements to fit a page

double a[N], b[N], temp;
for (kk=0; kk<N; kk+=NUMPERPAGE) {
    temp = a[kk+NUMPERPAGE];        // TLB priming for older archs
    // use block size = page size,
```

Example 9-11. A Memory Copy Routine Using Software Prefetch (Contd.)

```
    // prefetch entire block, one cache line per loop
    for (j=kk+16; j<kk+NUMPERPAGE; j+=16) {
      _mm_prefetch((char*)&a[j], _MM_HINT_NTA);
   }
// copy 128 byte per loop
      for (j=kk; j<kk+NUMPERPAGE; j+=16) {
     _mm_stream_ps((float*)&b[j],
                          _mm_load_ps((float*)&a[j]));
            _mm_stream_ps((float*)&b[j+2],
                          _mm_load_ps((float*)&a[j+2]));
            _mm_stream_ps((float*)&b[j+4],
                          _mm_load_ps((float*)&a[j+4]));
            _mm_stream_ps((float*)&b[j+6],
                          _mm_load_ps((float*)&a[j+6]));
            _mm_stream_ps((float*)&b[j+8],
                          _mm_load_ps((float*)&a[j+8]));
            _mm_stream_ps((float*)&b[j+10],
                          _mm_load_ps((float*)&a[j+10]));
            _mm_stream_ps((float*)&b[j+12],
                          _mm_load_ps((float*)&a[j+12]));
            _mm_stream_ps((float*)&b[j+14],
                          _mm_load_ps((float*)&a[j+14]));
      }  // finished copying one block
}           // finished copying N elements
_mm_sfence();
```

9.6.2.5 Using the 8-byte Streaming Stores and Software Prefetch

Example 9-11 presents the copy algorithm that uses second level cache. The algorithm performs the following steps:

1. Uses blocking technique to transfer 8-byte data from memory into second-level cache using the _MM_PREFETCH intrinsic, 128 bytes at a time to fill a block. The size of a block should be less than one half of the size of the second-level cache, but large enough to amortize the cost of the loop.
2. Loads the data into an XMM register using the _MM_LOAD_PS intrinsic.
3. Transfers the 8-byte data to a different memory location via the _MM_STREAM intrinsics, bypassing the cache.

In Example 9-11, eight _MM_LOAD_PS and _MM_STREAM_PS intrinsics are used so that all of the data prefetched (a 128-byte cache line) is written back. The prefetch and streaming-stores are executed in separate loops to minimize the number of transitions between reading and writing data. This significantly improves the bandwidth of the memory accesses.

The TEMP = A[KK+CACHESIZE] instruction is used to ensure the page table entry for array in older architectures, and A is entered in the TLB prior to prefetching. This is essentially a prefetch itself, as a cache line is filled from that memory location with this instruction. Hence, the prefetching starts from KK+4 in this loop.

This example assumes that the destination of the copy is not temporally adjacent to the code. If the copied data is destined to be reused in the near future, then the streaming store instructions should be replaced with regular 128 bit stores (_MM_STORE_PS).

9.6.2.6 Using 16-byte Streaming Stores and Hardware Prefetch

An alternate technique for optimizing a large region memory copy is to take advantage of hardware prefetcher, 16-byte streaming stores, and apply a segmented approach to separate bus read and write transactions. See Section 3.6.11

The technique employs two stages. In the first stage, a block of data is read from memory to the cache sub-system. In the second stage, cached data are written to their destination using streaming stores.

Example 9-12. Memory Copy Using Hardware Prefetch and Bus Segmentation

```
void block_prefetch(void *dst,void *src)
{ _asm {
        mov edi,dst
        mov esi,src
        mov edx,SIZE
        align 16
    main_loop:
        xor ecx,ecx
        align 16
}

    prefetch_loop:
        movaps xmm0, [esi+ecx]
        movaps xmm0, [esi+ecx+64]
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne prefetch_loop
        xor ecx,ecx
        align 16
        cpy_loop:
        movdqa xmm0,[esi+ecx]
        movdqa xmm1,[esi+ecx+16]
        movdqa xmm2,[esi+ecx+32]
        movdqa xmm3,[esi+ecx+48]
        movdqa xmm4,[esi+ecx+64]

        movdqa xmm5,[esi+ecx+16+64]
        movdqa xmm6,[esi+ecx+32+64]
        movdqa xmm7,[esi+ecx+48+64]
        movntdq [edi+ecx],xmm0
        movntdq [edi+ecx+16],xmm1
        movntdq [edi+ecx+32],xmm2
        movntdq [edi+ecx+48],xmm3
```

Example 9-12. Memory Copy Using Hardware Prefetch and Bus Segmentation (Contd.)

```
        movntdq [edi+ecx+64],xmm4
        movntdq [edi+ecx+80],xmm5
        movntdq [edi+ecx+96],xmm6
        movntdq [edi+ecx+112],xmm7
        add ecx,128
        cmp ecx,BLOCK_SIZE
        jne cpy_loop

        add esi,ecx
        add edi,ecx
        sub edx,ecx
        jnz main_loop
        sfence
    }
}
```

9.6.2.7 Performance Comparisons of Memory Copy Routines

The throughput of a large-region, memory copy routine depends on several factors:

- Coding techniques that implements the memory copy task.
- Characteristics of the system bus (speed, peak bandwidth, overhead in read/write transaction protocols).
- Microarchitecture of the processor.

The baseline for performance comparison is the throughput (bytes/sec) of 8-MByte region memory with a 400-MHz system bus using byte-sequential technique similar to that shown in Example 9-10. The degree of improvement relative to the performance baseline for some recent processors and platforms with higher system bus speed using different coding techniques are compared.

The second coding technique moves data at 4-Byte granularity using REP string instruction. The third column compares the performance of the coding technique listed in Example 9-11. The fourth column of performance compares the throughput of fetching 4-KBytes of data at a time (using hardware prefetch to aggregate bus read transactions) and writing to memory via 16-Byte streaming stores.

Increases in bus speed is the primary contributor to throughput improvements. The technique shown in Example 9-12 will likely take advantage of the faster bus speed in the platform more efficiently. Additionally, increasing the block size to multiples of 4-KBytes while keeping the total working set within the second-level cache can improve the throughput slightly.

The relative performance figure shown in Table 9-5 is representative of clean microarchitectural conditions within a processor (e.g. looping s simple sequence of code many times). The net benefit of integrating a specific memory copy routine into an application (full-featured applications tend to create many complicated micro-architectural conditions) will vary for each application.

9.6.3 DETERMINISTIC CACHE PARAMETERS

If CPUID supports the deterministic parameter leaf, software can use the leaf to query each level of the cache hierarchy. Enumeration of each cache level is by specifying an index value (starting form 0) in the ECX register (see "CPUID-CPU Identification" in Chapter 3, "Instruction Set Reference, A-L" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

The list of parameters is shown in Table 9-5.

Table 9-5. Deterministic Cache Parameters Leaf

Bit Location	Name	Meaning
EAX[4:0]	Cache Type	0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved
EAX[7:5]	Cache Level	Starts at 1
EAX[8]	Self Initializing cache level	1: does not need SW initialization
EAX[9]	Fully Associative cache	1: Yes
EAX[13:10]	Reserved	-
EAX[25:14]	Maximum number of logical processors sharing this cache	Plus 1 encoding
EAX[31:26]	Maximum number of cores in a package	Plus 1 encoding
EBX[11:0]	System Coherency Line Size (L)	Plus 1 encoding (Bytes)
EBX[21:12]	Physical Line partitions (P)	Plus 1 encoding
EBX[31:22]	Ways of associativity (W)	Plus 1 encoding
ECX[31:0]	Number of Sets (S)	Plus 1 encoding
EDX	Reserved	-
CPUID leaves > 3 < 80000000 are only visible when IA32_CR_MISC_ENABLES.BOOT_NT4 (bit 22) is clear (Default).		

The deterministic cache parameter leaf provides a means to implement software with a degree of forward compatibility with respect to enumerating cache parameters. Deterministic cache parameters can be used in several situations, including:

- Determine the size of a cache level.
- Adapt cache blocking parameters to different sharing topology of a cache-level across Intel HT Technology, multicore and single-core processors.
- Determine multithreading resource topology in an MP system..
- Determine cache hierarchy topology in a platform using multicore processors.
- Manage threads and processor affinities.
- Determine prefetch stride.

The size of a given level of cache is given by:

(# of Ways) * (Partitions) * (Line_size) * (Sets) = (EBX[31:22] + 1) * (EBX[21:12] + 1) * (EBX[11:0] + 1) * (ECX + 1)

9.6.3.1 Cache Sharing Using Deterministic Cache Parameters

Improving cache locality is an important part of software optimization. For example, a cache blocking algorithm can be designed to optimize block size at runtime for single-processor implementations and a variety of multiprocessor execution environments (including processors supporting Intel HT Technology, or multicore processors).

The basic technique is to place an upper limit of the blocksize to be less than the size of the target cache level divided by the number of logical processors serviced by the target level of cache. This technique is applicable to multithreaded application programming. The technique can also benefit single-threaded applications that are part of a multi-tasking workloads.

9.6.3.2 Cache Sharing in Single-Core or Multicore

Deterministic cache parameters are useful for managing shared cache hierarchy in multithreaded applications for more sophisticated situations. A given cache level may be shared by logical processors in a processor or it may be implemented to be shared by logical processors in a physical processor package.

Using the deterministic cache parameter leaf and initial APIC_ID associated with each logical processor in the platform, software can extract information on the number and the topological relationship of logical processors sharing a cache level.

9.6.3.3 Determine Prefetch Stride

The prefetch stride (see description of CPUID.01H.EBX) provides the length of the region that the processor will prefetch with the PREFETCHh instructions (PREFETCHT0, PREFETCHT1, PREFETCHT2 and PREFETCHNTA). Software will use the length as the stride when prefetching into a particular level of the cache hierarchy as identified by the instruction used. The prefetch size is relevant for cache types of Data Cache (1) and Unified Cache (3); it should be ignored for other cache types. Software should not assume that the coherency line size is the prefetch stride.

If the prefetch stride field is zero, then software should assume a default size of 64 bytes is the prefetch stride. Software should use the following algorithm to determine what prefetch size to use depending on whether the deterministic cache parameter mechanism is supported or the legacy mechanism:

- If a processor supports the deterministic cache parameters and provides a non-zero prefetch size, then that prefetch size is used.
- If a processor supports the deterministic cache parameters and does not provides a prefetch size then default size for each level of the cache hierarchy is 64 bytes.
- If a processor does not support the deterministic cache parameters but provides a legacy prefetch size descriptor (0xF0 - 64 byte, 0xF1 - 128 byte) will be the prefetch size for all levels of the cache hierarchy.
- If a processor does not support the deterministic cache parameters and does not provide a legacy prefetch size descriptor, then 32-bytes is the default size for all levels of the cache hierarchy.

CHAPTER 10
SUB-NUMA CLUSTERING

Sub-NUMA Clustering (SNC) is a mode for improving average latency from last level cache (LLC) to local memory. It replaces the Cluster-on-Die (COD) implementation which was used in the previous generation of the Intel® Xeon® processor E5 family.

10.1 SUB-NUMA CLUSTERING

SNC can improve the average LLC/memory latency by splitting the LLC into disjoint clusters based on address range, with each cluster bound to a subset of memory controllers in the system.

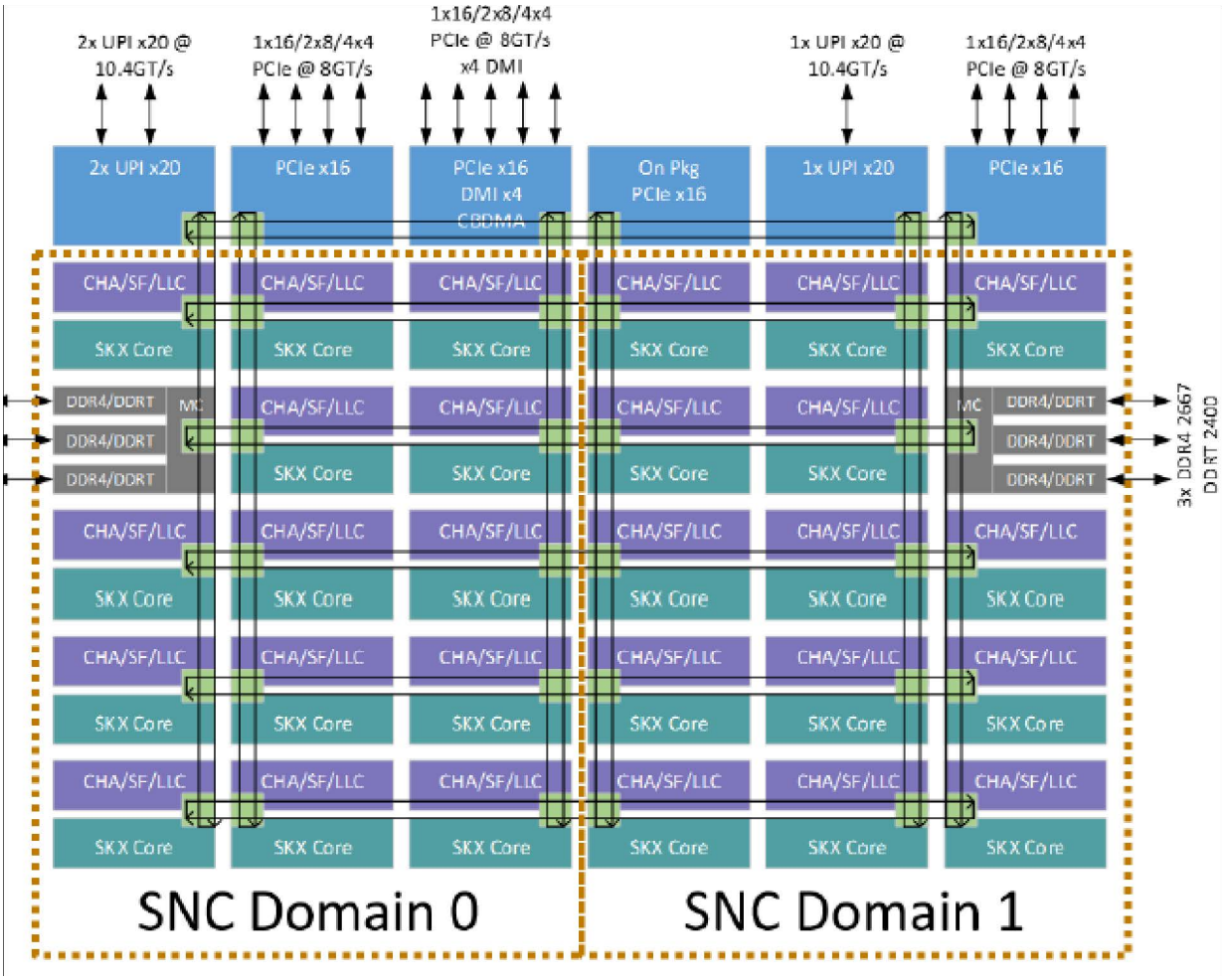


Figure 10-1. Example of SNC Configuration

10.2 COMPARISON WITH CLUSTER-ON-DIE

SNC provides similar localization benefits to those of COD, but without some of COD's disadvantages. Unlike COD, SNC has the following properties.

- Only one Ultra Path Interconnect (UPI) caching agent is required.
- Memory access latency in remote clusters is smaller, as no UPI flow is needed.
- It uses LLC capacity more efficiently as there is no duplication of lines in the LLC.

A disadvantage of SNC is listed below.

- Remote cluster addresses are never cached in local cluster LLC, resulting in larger latency compared to Cluster-on-Die (COD) in some cases.

10.3 SNC USAGE

This section describes the following modes and their BIOS names in brackets (the exact BIOS parameter names may vary depending on the BIOS vendor and version).

- NUMA disabled (NUMA Optimized: Disabled)
- SNC off (Integrated Memory Controller (IMC) Interleaving: auto, NUMA Optimized: Enabled, Sub-NUMA Cluster: Disabled)
- SNC on (IMC Interleaving: 1-way Interleave, NUMA Optimized: Enabled, Sub_NUMA Cluster: Enabled)

The commands that follow were executed on a 2-socket Intel® Xeon® system, 28 cores per a socket, Intel® Hyper-Threading Technology (Intel® HT) enabled.

10.3.1 HOW TO CHECK NUMA CONFIGURATION

There are additional NUMA nodes in a system with SNC enabled; to get benefits from the SNC feature, a developer should be aware of the NUMA configuration.

This section describes different ways to check NUMA system configuration.

libnuma

An application can check NUMA configuration with **libnuma**.

As an example this code uses the `libnuma` library to find the maximum number of NUMA nodes.

Example 10-1. Code Using libnuma to Find the Maximum Number of NUMA Node

```
#include <stdio.h>
#include <stdlib.h>
#include <numa.h>

int main(int argc, char *argv[])
```

Example 10-1. (Contd.)Code Using libnuma to Find the Maximum Number of NUMA Node

```
{ int max_node;

/* Check the system for NUMA support */
   max_node = numa_max_node();
   printf("%d\n", max_node);
return 0;
}

numactl
//In Linux* you can check the NUMA configuration with the numactl utility (the numactl-libs, and numactl-devel packages might also be required).
$ numactl --hardware

NUMA disabled:
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 0 size: 196045 MB
node 0 free: 190581 MB
node distances:
node   0
  0:  10

SNC off:
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
node 0 size: 96973 MB

node 0 free: 94089 MB
node 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 1 size: 98304 MB
node 1 free: 95694 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10

node 0 size: 47821 MB
node 0 free: 45759 MB
node 1 cpus: 4 5 6 10 11 12 13 18 19 20 24 25 26 27 60 61 62 66 67 68 69 74 75 76 80 81 82 83
node 1 size: 49152 MB
node 1 free: 47097 MB
node 2 cpus: 28 29 30 31 35 36 37 42 43 44 45 49 50 51 84 85 86 87 91 92 93 98 99 100 101 105 106 107
node 2 size: 49152 MB
node 2 free: 47617 MB
```

Example 10-1. (Contd.)Code Using libnuma to Find the Maximum Number of NUMA Node

```
node 3 cpus: 32 33 34 38 39 40 41 46 47 48 52 53 54 55 88 89 90 94 95 96 97 102 103 104 108 109 110 111
node 3 size: 49152 MB
node 3 free: 47231 MB
node distances:
node   0   1   2   3
  0:  10  11  21  21
  1:  11  10  21  21
  2:  21  21  10  11
3:   21   21   11   10
SNC on:
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 7 8 9 14 15 16 17 21 22 23 56 57 58 59 63 64 65 70 71 72 73 77 78 79
```

hwloc

In Linux* you can also check the NUMA configuration with the `lstopo` utility (the `hwloc` package is required). For example:

$ lstopo -p --of png --no-io --no-caches > numa_topology.png

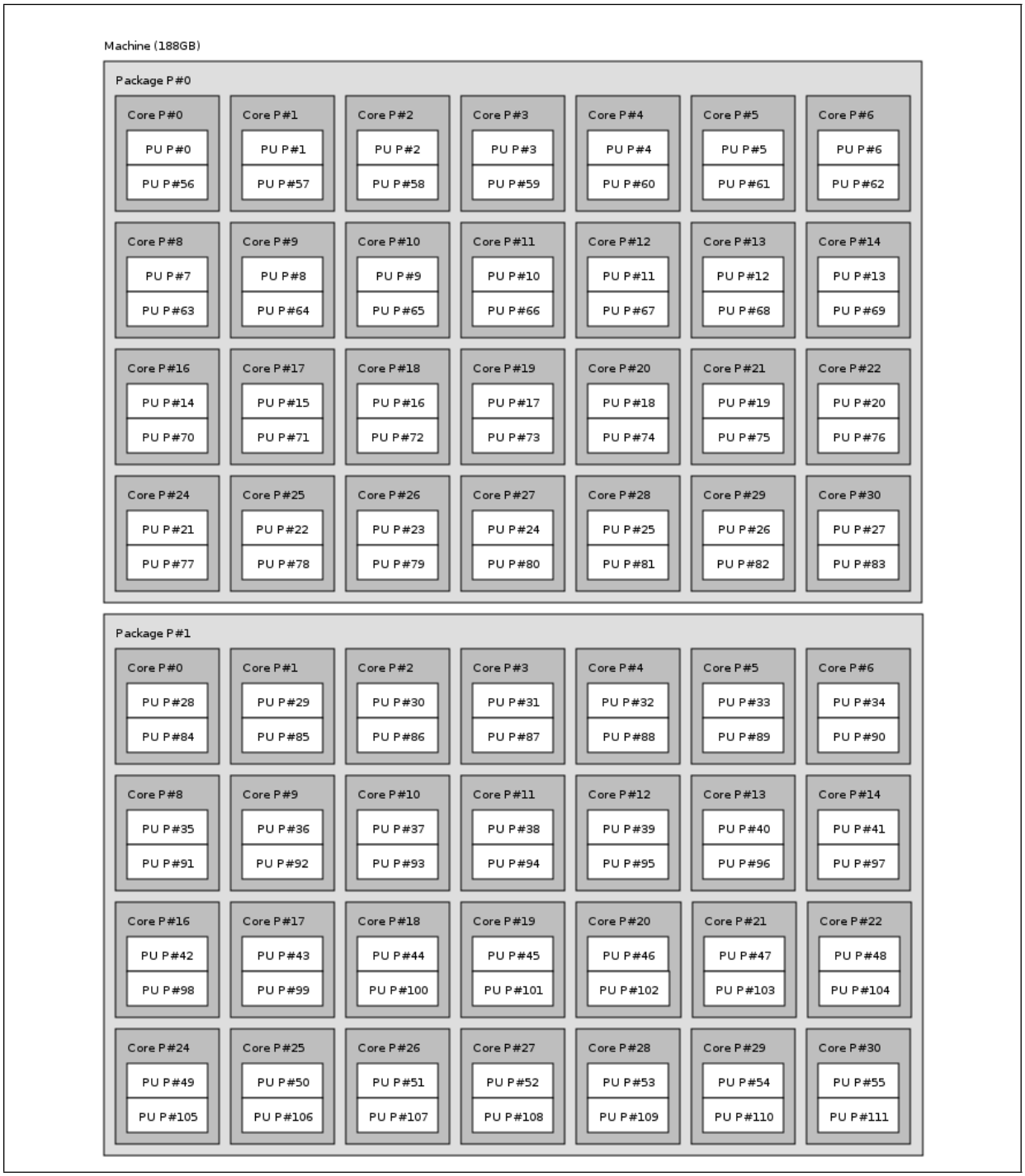


Figure 10-2. NUMA Disabled

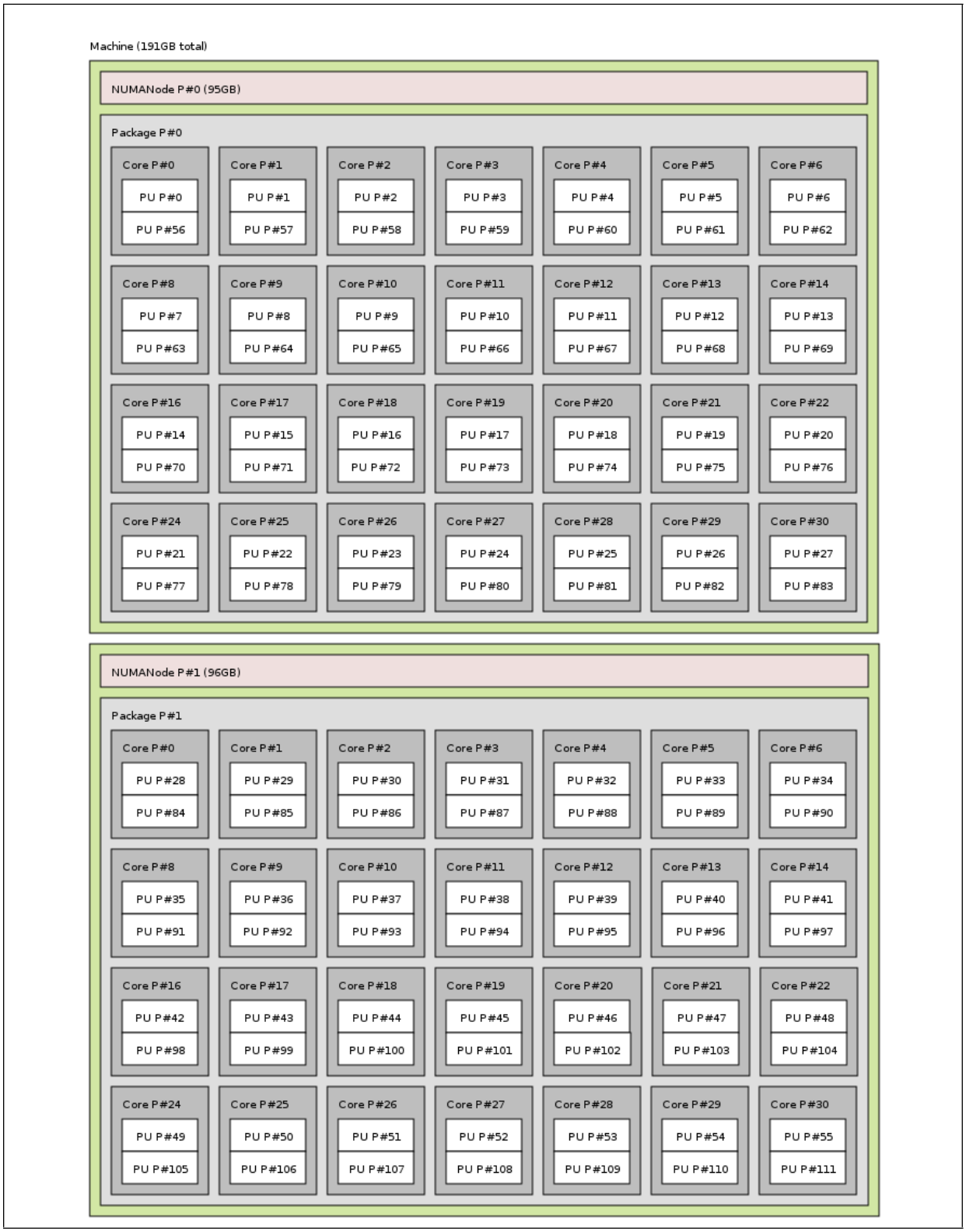


Figure 10-3. SNC Off

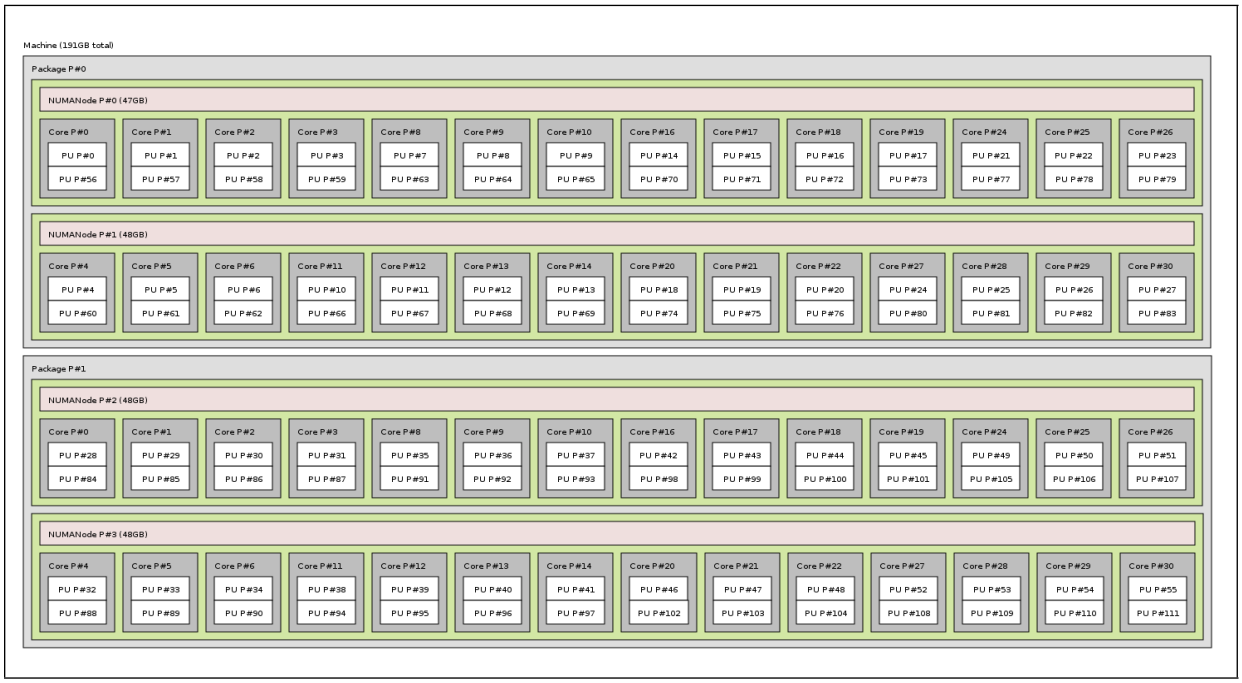


Figure 10-4. SNC On

10.3.2 MPI OPTIMIZATIONS FOR SNC

Software needs to be NUMA optimized to benefit from SNC. Running one MPI rank per NUMA region trivially ensures locality-of-access without requiring changes to the code to ensure that it behaves in a NUMA friendly manner. This is a simple way to improve performance through the use of SNC.

The Intel® MPI Library includes some NUMA-related optimizations. The out-of-the-box behavior of the Intel MPI Library should cover most cases, but there are some environment variables available to control NUMA-related features that can improve performance in specific cases.

The relevant environment variables mainly relate to MPI process placement, that is, process pinning/binding – such as the I_MPI_PIN_DOMAIN variable. For more information, see the Intel® MPI Library Developer Reference. This environment variable defines some non-overlapping subsets (domains) of logical processors on a node, and a set of rules for how MPI processes are bound to these domains: one MPI process per domain, as illustrated below.

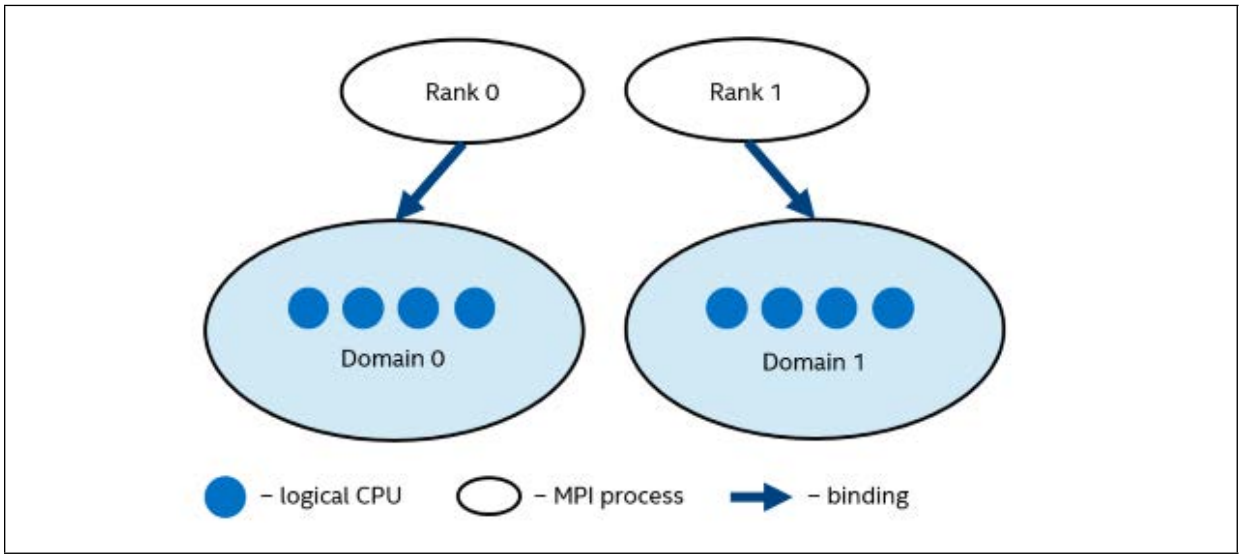


Figure 10-5. Domain Example with One MPI Process Per Domain

Each MPI process can create some child threads to run within the corresponding domain. The process' threads can freely migrate from one logical processor to another within the particular domain.

For example, `I_MPI_PIN_DOMAIN=numa` may be a reasonable option for hybrid MPI/OpenMP* applications with SNC mode enabled. In this case, each domain consists of logical processors that share a particular NUMA node. The number of domains on a machine is equal to the number of NUMA nodes on the machine.

Please see the Intel MPI Library documentation for detailed information.

10.3.3 SNC PERFORMANCE COMPARISON

This section contains performance data collected with Intel® Memory Latency Checker (Intel® MLC) to demonstrate the variations in performance (latency) between NUMA nodes in different modes.

An important factor in determining application performance is the time required for the application to fetch data from the processor's cache hierarchy and from the memory subsystem. Local memory and cross-socket memory latencies vary significantly in a NUMA-enabled multi-socket system. Bandwidth also plays an important role in determining performance. So measuring these latencies and bandwidths is important when establishing a baseline for the system being tested, and performing performance analysis.

Intel MLC is a tool used to measure memory latencies and bandwidth, and how they change as the load on the system increases. It also provides several options for more fine-grained investigation where bandwidth and latencies from a specific set of cores to caches or memory can be measured as well.

For details, see Intel® Memory Latency Checker v.3.10 (Intel® MLC).

The following command was used to collect the performance data:

% mlc_avx512 --latency_matrix

This command measures idle memory latency from each socket in the system to every other socket and reports the results in a matrix. The default invocation reports latencies to all of the NUMA nodes in the system. NUMA-level reporting works only on Linux. On Windows, only socket level reporting is supported.

NOTE

It is challenging to measure memory latencies on modern Intel processors accurately as they have sophisticated HW prefetchers. Intel MLC automatically disables these prefetchers while measuring the latencies and restores them to their previous state on completion. The prefetcher control is exposed through an MSR and MSR access requires root level permission. So, Intel MLC needs to be run as 'root' on Linux.

The software configuration used for these measurements is Intel MLC v3.3-Beta2, Red Hat* Linux* 7.2.

NUMA disabled:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Memory node	
Socket	0	1
0	126.5	129.4
1	123.1	122.6

SNC off:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Numa node	
Numa node	0	1
0	81.9	153.1
1	153.7	82.0

SNC on:

Using buffer size of 2000.000MB

Measuring idle latencies (in ns)...

	Numa node			
Numa node	0	1	2	3
0	81.6	89.4	140.4	153.6
1	86.5	78.5	144.3	162.8
2	142.3	153.0	81.6	89.3
3	144.5	162.8	85.5	77.4

CHAPTER 11
MULTICORE AND INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

This chapter describes software optimization techniques for multithreaded applications running in an environment using either multiprocessor (MP) systems or processors with hardware-based multithreading support. Multiprocessor systems are systems with two or more sockets, each mated with a physical processor package. Intel® 64 and IA-32 processors that provide hardware multithreading support include dual-core processors, quad-core processors and processors supporting Intel® Hyper-Threading Technology (Intel® HT Technology)[1].

Computational throughput in a multithreading environment can increase as more hardware resources are added to take advantage of thread-level or task-level parallelism. Hardware resources can be added in the form of more than one physical-processor, processor-core-per-package, and/or logical-processor-per-core. Therefore, there are some aspects of multithreading optimization that apply across MP, multicore, and Intel HT Technology. There are also some specific microarchitectural resources that may be implemented differently in different hardware multithreading configurations (for example: execution resources are not shared across different cores but shared by two logical processors in the same core if HT Technology is enabled). This chapter covers guidelines that apply to these situations.

This chapter covers:

- Performance characteristics and usage models.
- Programming models for multithreaded applications.
- Software optimization techniques in five specific areas.

11.1 PERFORMANCE AND USAGE MODELS

The performance gains of using multiple processors, multicore processors or Intel HT Technology are greatly affected by the usage model and the amount of parallelism in the control flow of the workload. Two common usage models are:

- Multithreaded applications.
- Multitasking using single-threaded applications.

11.1.1 MULTITHREADING

When an application employs multithreading to exploit task-level parallelism in a workload, the control flow of the multi-threaded software can be divided into two parts: parallel tasks and sequential tasks.

Amdahl's Law describes an application's performance gain as it relates to the degree of parallelism in the control flow. It is a useful guide for selecting the code modules, functions, or instruction sequences that are most likely to realize the most gains from transforming sequential tasks and control flows into parallel code to take advantage multithreading hardware support.

---

1. The presence of hardware multithreading support in Intel 64 and IA-32 processors can be detected by checking the feature flag CPUID .01H:EDX[28]. A return value of in bit 28 indicates that at least one form of hardware multithreading is present in the physical processor package. The number of logical processors present in each package can also be obtained from CPUID. The application must check how many logical processors are enabled and made available to application at runtime by making the appropriate operating system calls. See the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for information.

Figure 11-1 illustrates how performance gains can be realized for any workload according to Amdahl's Law. The bar in Figure 11-1 represents an individual task unit or the collective workload of an entire application.

In general, the speed-up of running multiple threads on an MP systems with *N* physical processors, over single-threaded execution, can be expressed as:

$$\text{RelativeResponse} = \frac{\text{Tsequential}}{\text{Tparallel}} = \left(1 - P + \frac{P}{N} + O\right)$$

where *P* is the fraction of workload that can be parallelized, and *O* represents the overhead of multithreading and may vary between different operating systems. In this case, performance gain is the inverse of the relative response.

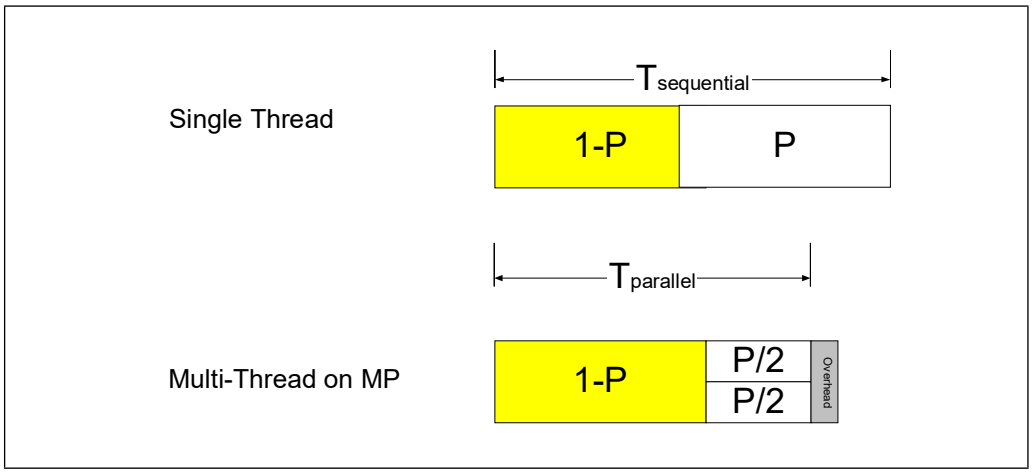


Figure 11-1. Amdahl's Law and MP Speed-up

When optimizing application performance in a multithreaded environment, control flow parallelism is likely to have the largest impact on performance scaling with respect to the number of physical processors and to the number of logical processors per physical processor.

If the control flow of a multi-threaded application contains a workload in which only 50% can be executed in parallel, the maximum performance gain using two physical processors is only 33%, compared to using a single processor. Using four processors can deliver no more than a 60% speed-up over a single processor. Thus, it is critical to maximize the portion of control flow that can take advantage of parallelism. Improper implementation of thread synchronization can significantly increase the proportion of serial control flow and further reduce the application's performance scaling.

In addition to maximizing the parallelism of control flows, interaction between threads in the form of thread synchronization and imbalance of task scheduling can also impact overall processor scaling significantly.

Excessive cache misses are one cause of poor performance scaling. In a multithreaded execution environment, they can occur from:

- Aliased stack accesses by different threads in the same process.
- Thread contentions resulting in cache line evictions.
- False-sharing of cache lines between different processors.

Techniques that address each of these situations (and many other areas) are described in sections in this chapter.

11.1.2 MULTITASKING ENVIRONMENT

Hardware multithreading capabilities in Intel 64 and IA-32 processors can exploit task-level parallelism when a workload consists of several single-threaded applications and these applications are scheduled to run concurrently under an MP-aware operating system. In this environment, hardware multithreading capabilities can deliver higher throughput for the workload, although the relative performance of a single task (in terms of time of completion relative to the same task when in a single-threaded environment) will vary, depending on how much shared execution resources and memory are utilized.

For development purposes, several popular operating systems (for example Microsoft Windows* XP Professional and Home, Linux* distributions using kernel 2.4.19 or later[1]) include OS kernel code that can manage the task scheduling and the balancing of shared execution resources within each physical processor to maximize the throughput.

Because applications run independently under a multitasking environment, thread synchronization issues are less likely to limit the scaling of throughput. This is because the control flow of the workload is likely to be 100% parallel[2] (if no inter-processor communication is taking place and if there are no system bus constraints).

With a multitasking workload, however, bus activities and cache access patterns are likely to affect the scaling of the throughput. Running two copies of the same application or same suite of applications in a lock-step can expose an artifact in performance measuring methodology. This is because an access pattern to the first level data cache can lead to excessive cache misses and produce skewed performance results. Fix this problem by:

- Including a per-instance offset at the start-up of an application.
- Introducing heterogeneity in the workload by using different datasets with each instance of the application.
- Randomizing the sequence of start-up of applications when running multiple copies of the same suite.

When two applications are employed as part of a multitasking workload, there is little synchronization overhead between these two processes. It is also important to ensure each application has minimal synchronization overhead within itself.

An application that uses lengthy spin loops for intra-process synchronization is less likely to benefit from HT Technology in a multitasking workload. This is because critical resources will be consumed by the long spin loops.

11.2 PROGRAMMING MODELS AND MULTITHREADING

Parallelism is the most important concept in designing a multithreaded application and realizing optimal performance scaling with multiple processors. An optimized multithreaded application is characterized by large degrees of parallelism or minimal dependencies in the following areas:

- Workload.
- Thread interaction.
- Hardware utilization.

The key to maximizing workload parallelism is to identify multiple tasks that have minimal inter-dependencies within an application and to create separate threads for parallel execution of those tasks.

Concurrent execution of independent threads is the essence of deploying a multithreaded application on a multiprocessing system. Managing the interaction between threads to minimize the cost of thread synchronization is also critical to achieving optimal performance scaling with multiple processors.

---

1. This code is included in Red Hat* Linux Enterprise AS 2.1.
2. A software tool that attempts to measure the throughput of a multitasking workload is likely to introduce control flows that are not parallel. Thread synchronization issues must be considered as an integral part of its performance measuring methodology.

Efficient use of hardware resources between concurrent threads requires optimization techniques in specific areas to prevent contentions of hardware resources. Coding techniques for optimizing thread synchronization and managing other hardware resources are discussed in subsequent sections.

Parallel programming models are discussed next.

11.2.1 PARALLEL PROGRAMMING MODELS

Two common programming models for transforming independent task requirements into application threads are:

- Domain decomposition.
- Functional decomposition.

11.2.1.1 Domain Decomposition

Usually large compute-intensive tasks use data sets that can be divided into several small subsets, each having a large degree of computational independence. Examples include:

- Computation of a discrete cosine transformation (DCT) on two-dimensional data by dividing the two-dimensional data into several subsets and creating threads to compute the transform on each subset.
- Matrix multiplication; here, threads can be created to handle the multiplication of half of matrix with the multiplier matrix.

Domain Decomposition is a programming model based on creating identical or similar threads to process smaller pieces of data independently. This model can take advantage of duplicated execution resources present in a traditional multiprocessor system. It can also take advantage of shared execution resources between two logical processors in Intel HT Technology. This is because a data domain thread typically consumes only a fraction of the available on-chip execution resources.

Section 11.3.4 discusses additional guidelines that can help data domain threads use shared execution resources cooperatively and avoid the pitfalls creating contentions of hardware resources between two threads.

11.2.2 FUNCTIONAL DECOMPOSITION

Applications usually process a wide variety of tasks with diverse functions and many unrelated data sets. For example, a video codec needs several different processing functions. These include DCT, motion estimation and color conversion. Using a functional threading model, applications can program separate threads to do motion estimation, color conversion, and other functional tasks.

Functional decomposition will achieve more flexible thread-level parallelism if it is less dependent on the duplication of hardware resources. For example, a thread executing a sorting algorithm and a thread executing a matrix multiplication routine are not likely to require the same execution unit at the same time. A design recognizing this could advantage of traditional multiprocessor systems as well as multiprocessor systems using processors supporting Intel HT Technology.

11.2.3 SPECIALIZED PROGRAMMING MODELS

Intel Core Duo processor and processors based on Intel Core microarchitecture offer a second-level cache shared by two processor cores in the same physical package. This provides opportunities for two application threads to access some application data while minimizing the overhead of bus traffic.

Multi-threaded applications may need to employ specialized programming models to take advantage of this type of hardware feature. One such scenario is referred to as producer-consumer. In this scenario, one thread writes data into

some destination (hopefully in the second-level cache) and another thread executing on the other core in the same physical package subsequently reads data produced by the first thread.

The basic approach for implementing a producer-consumer model is to create two threads; one thread is the producer and the other is the consumer. Typically, the producer and consumer take turns to work on a buffer and inform each other when they are ready to exchange buffers. In a producer-consumer model, there is some thread synchronization overhead when buffers are exchanged between the producer and consumer. To achieve optimal scaling with the number of cores, the synchronization overhead must be kept low. This can be done by ensuring the producer and consumer threads have comparable time constants for completing each incremental task prior to exchanging buffers.

Example 11-1 illustrates the coding structure of single-threaded execution of a sequence of task units, where each task unit (either the producer or consumer) executes serially (shown in Figure 11-2). In the equivalent scenario under multi-threaded execution, each producer-consumer pair is wrapped as a thread function and two threads can be scheduled on available processor resources simultaneously.

Example 11-1. Serial Execution of Producer and Consumer Work Items

Table 1-1

```
for (i = 0; i < number_of_iterations; i++) {
    producer (i, buff);  // pass buffer index and buffer address
    consumer (i, buff);
}(
```

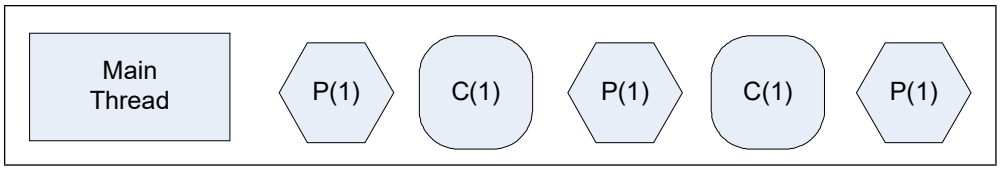


Figure 11-2. Single-threaded Execution of Producer-consumer Threading Model

11.2.3.1 Producer-Consumer Threading Models

Figure 11-3 illustrates the basic scheme of interaction between a pair of producer and consumer threads. The horizontal direction represents time. Each block represents a task unit, processing the buffer assigned to a thread.

The gap between each task represents synchronization overhead. The decimal number in the parenthesis represents a buffer index. On an Intel Core Duo processor, the producer thread can store data in the second-level cache to allow the consumer thread to continue work requiring minimal bus traffic.

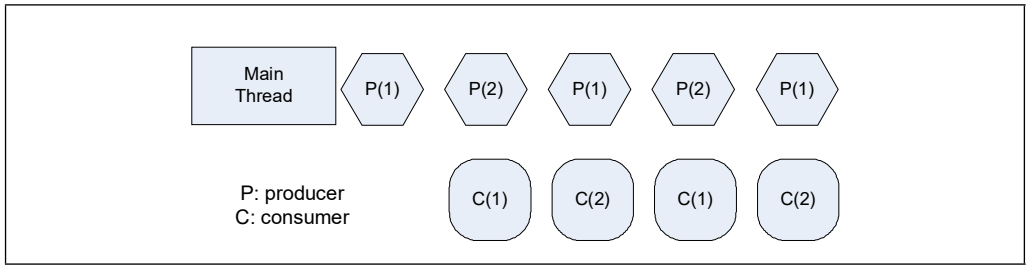


Figure 11-3. Execution of Producer-consumer Threading Model on a Multicore Processor

The basic structure to implement the producer and consumer thread functions with synchronization to communicate buffer index is shown in Example 11-2.

Example 11-2. Basic Structure of Implementing Producer Consumer Threads

```
 (a) Basic structure of a producer thread function
void producer_thread()
{    int iter_num = workamount - 1; // make local copy
     int mode1 = 1; // track usage of two buffers via 0 and 1
     produce(buffs[0],count);  // placeholder function
     while (iter_num--)  {
Signal(&signal1,1);  // tell the other thread to commence
          produce(buffs[mode1],count); // placeholder function
          WaitForSignal(&end1);
          mode1 = 1 - mode1;  // switch to the other buffer
     }
}
b) Basic structure of a consumer thread
void consumer_thread()
{    int mode2 = 0;  // first iteration start with buffer 0, than alternate
     int iter_num = workamount - 1;
     while (iter_num--)  {
     WaitForSignal(&signal1);
          consume(buffs[mode2],count); // placeholder function
          Signal(&end1,1);
          mode2 = 1 - mode2;
     }
     consume(buffs[mode2],count);
}
```

It is possible to structure the producer-consumer model in an interlaced manner so it can minimize bus traffic and be effective on multicore processors without shared second-level cache.

In this interlaced variation of the producer-consumer model, each scheduling quanta of an application thread comprises of a producer task and a consumer task. Two identical threads are created to execute in parallel. During each scheduling quanta of a thread, the producer task starts first and the consumer task follows after the completion of the producer task; both tasks work on the same buffer. As each task completes, one thread signals to the other thread notifying its corresponding task to use its designated buffer. Thus, the producer and consumer tasks execute in parallel in two threads. As long as the data generated by the producer reside in either the first or second level cache of the same core, the consumer can access them without incurring bus traffic. The scheduling of the interlaced producer-consumer model is shown in Figure 11-4.

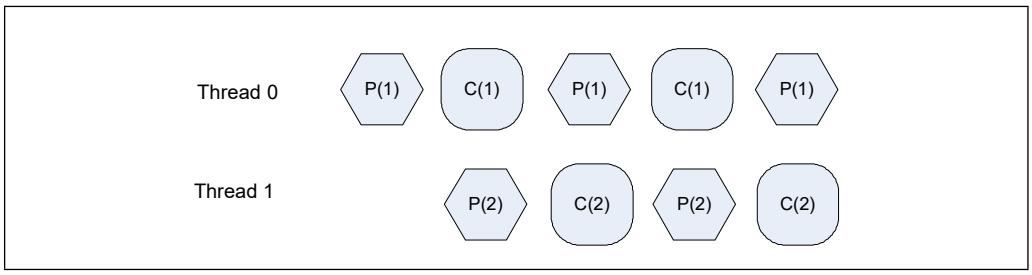


Figure 11-4. Interlaced Variation of the Producer Consumer Model

Example 11-3 shows the basic structure of a thread function that can be used in this interlaced producer-consumer model.

Example 11-3. Thread Function for an Interlaced Producer Consumer Model

```
// master thread starts first iteration, other thread must wait
// one iteration
void producer_consumer_thread(int master)
{
int mode = 1 - master; // track which thread and its designated
                                // buffer index
unsigned int iter_num = workamount >> 1;
unsigned int i=0;

iter_num += master & workamount & 1;
    if (master)  // master thread starts the first iteration
    {
          produce(buffs[mode],count);
          Signal(sigp[1-mode1],1);  // notify producer task in follower
                                          // thread that it can proceed
     consume(buffs[mode],count);
          Signal(sigc[1-mode],1);
     i = 1;
    }

for (; i < iter_num; i++)
    {
     WaitForSignal(sigp[mode]);
          produce(buffs[mode],count); // notify the producer task in
                                                 // other thread
          Signal(sigp[1-mode],1);
WaitForSignal(sigc[mode]);
     consume(buffs[mode],count);
          Signal(sigc[1-mode],1);
    }
}
```

11.2.4 TOOLS FOR CREATING MULTITHREADED APPLICATIONS

Programming directly to a multithreading application programming interface (API) is not the only method for creating multithreaded applications. New tools (such as the Intel compiler) have become available with capabilities that make the challenge of creating multithreaded application easier.

Features available in the latest Intel compilers are:

- Generating multithreaded code using OpenMP* directives[1].
- Generating multithreaded code automatically from unmodified high-level code[2].

11.2.4.1 Programming with OpenMP Directives

OpenMP provides a standardized, non-proprietary, portable set of Fortran and C++ compiler directives supporting shared memory parallelism in applications. OpenMP supports directive-based processing. This uses special preprocessors or modified compilers to interpret parallelism expressed in Fortran comments or C/C++ pragmas. Benefits of directive-based processing include:

- The original source can be compiled unmodified.
- It is possible to make incremental code changes. This preserves algorithms in the original code and enables rapid debugging.
- Incremental code changes help programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code.
- Offering directives to fine tune thread scheduling imbalance.
- Intel's implementation of OpenMP runtime can add minimal threading overhead relative to hand-coded multithreading.

11.2.4.2 Automatic Parallelization of Code

While OpenMP directives allow programmers to quickly transform serial applications into parallel applications, programmers must identify specific portions of the application code that contain parallelism and add compiler directives. Intel Compiler 6.0 supports a new (-QPARALLEL) option, which can identify loop structures that contain parallelism. During program compilation, the compiler automatically attempts to decompose the parallelism into threads for parallel processing. No other intervention or programmer is needed.

11.2.4.3 Supporting Development Tools

See Appendix A, "Application Performance Tools" for information on the various tools that Intel provides for software development.

11.3 OPTIMIZATION GUIDELINES

This section summarizes optimization guidelines for tuning multithreaded applications. Five areas are listed (in order of importance):

- Thread synchronization.
- Bus utilization.
- Memory optimization.

---

1. Intel Compiler 5.0 and later supports OpenMP directives. Visit http://software.intel.com for details.
2. Intel Compiler 6.0 supports auto-parallelization.

- Front end optimization.
- Execution resource optimization.

Practices associated with each area are listed in this section. Guidelines for each area are discussed in greater depth in sections that follow.

Most of the coding recommendations improve performance scaling with processor cores; and scaling due to HT Technology. Techniques that apply to only one environment are noted.

11.3.1 KEY PRACTICES OF THREAD SYNCHRONIZATION

Key practices for minimizing the cost of thread synchronization are summarized below:

- Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.
- Replace a spin-lock that may be acquired by multiple threads with pipelined locks so no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to acquire a lock.
- Use a thread-blocking API in a long idle loop to free up the processor.
- Prevent “false-sharing” of per-thread-data between two threads.
- Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.

See Section 11.4 for details.

11.3.2 KEY PRACTICES OF SYSTEM BUS OPTIMIZATION

Managing bus traffic can significantly impact the overall performance of multithreaded software and MP systems. Key practices of system bus optimization for achieving high data throughput and quick response are:

- Improve data and code locality to conserve bus command bandwidth.
- Avoid excessive use of software prefetch instructions and allow the automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.
- Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.
- Use full write transactions to achieve higher data throughput.

See Section 11.5 for details.

11.3.3 KEY PRACTICES OF MEMORY OPTIMIZATION

Key practices for optimizing memory operations are summarized below:

- Use cache blocking to improve locality of data access. Target one quarter to one half of cache size when targeting processors supporting HT Technology.
- Minimize the sharing of data between threads that execute on different physical processors sharing a common bus.
- Minimize data access patterns that are offset by multiples of 64-KBytes in each thread.
- Adjust the private stack of each thread in an application so the spacing between these stacks is not offset by multiples of 64 KBytes or 1 MByte (prevents unnecessary cache line evictions) when targeting processors supporting HT Technology.
- Add a per-instance stack offset when two instances of the same application are executing in lock steps to avoid memory accesses that are offset by multiples of 64 KByte or 1 MByte when targeting processors supporting HT Technology.

See Section 11.6 for details.

11.3.4 KEY PRACTICES OF EXECUTION RESOURCE OPTIMIZATION

Each physical processor has dedicated execution resources. Logical processors in physical processors supporting HT Technology share specific on-chip execution resources. Key practices for execution resource optimization include:

- Optimize each thread to achieve optimal frequency scaling first.
- Optimize multithreaded applications to achieve optimal scaling with respect to the number of physical processors.
- Use on-chip execution resources cooperatively if two threads are sharing the execution resources in the same physical processor package.
- For each processor supporting HT Technology, consider adding functionally uncorrelated threads to increase the hardware resource utilization of each physical processor package.

See Section 11.8 for details.

11.3.5 GENERALITY AND PERFORMANCE IMPACT

The next five sections cover the optimization techniques in detail. Recommendations discussed in each section are ranked by importance in terms of estimated local impact and generality.

Rankings are subjective and approximate. They can vary depending on coding style, application and threading domain. The purpose of including high, medium and low impact ranking with each recommendation is to provide a relative indicator as to the degree of performance gain that can be expected when a recommendation is implemented.

It is not possible to predict the likelihood of a code instance across many applications, so an impact ranking cannot be directly correlated to application-level performance gain. The ranking on generality is also subjective and approximate.

Coding recommendations that do not impact all three scaling factors are typically categorized as medium or lower.

11.4 THREAD SYNCHRONIZATION

Applications with multiple threads use synchronization techniques to ensure correct operation. However, thread synchronization that are improperly implemented can significantly reduce performance.

The best practice to reduce the overhead of thread synchronization is to start by reducing the application's requirements for synchronization. Intel Thread Profiler can be used to profile the execution timeline of each thread and detect situations where performance is impacted by frequent occurrences of synchronization overhead.

Several coding techniques and operating system (OS) calls are frequently used for thread synchronization. These include spin-wait loops, spin-locks, critical sections, to name a few. Choosing the optimal OS call for the circumstance and implementing synchronization code with parallelism in mind are critical in minimizing the cost of handling thread synchronization.

SSE3 provides two instructions (MONITOR/MWAIT) to help multithreaded software improve synchronization between multiple agents. In the first implementation of MONITOR and MWAIT, these instructions are available to operating system so that operating system can optimize thread synchronization in different areas. For example, an operating system can use MONITOR and MWAIT in its system idle loop (known as C0 loop) to reduce power consumption. An operating system can also use MONITOR and MWAIT to implement its C1 loop to improve the responsiveness of the C1 loop. See Chapter 9, "Multiple-Processor Management" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

11.4.1 CHOICE OF SYNCHRONIZATION PRIMITIVES

Thread synchronization often involves modifying some shared data while protecting the operation using synchronization primitives. There are many primitives to choose from. Guidelines that are useful when selecting synchronization primitives are:

- Favor compiler intrinsics or an OS provided interlocked API for atomic updates of simple data operation, such as increment and compare/exchange. This will be more efficient than other more complicated synchronization primitives with higher overhead.

  For more information on using different synchronization primitives, see the white paper, *Developing Multi-threaded Applications: A Platform Consistent Approach*.

- When choosing between different primitives to implement a synchronization construct, using Intel Thread Checker and Thread Profiler can be very useful in dealing with multithreading functional correctness issue and performance impact under multi-threaded execution. Additional information on the capabilities of Intel Thread Checker and Thread Profiler are described in Appendix A, "Application Performance Tools".

Table 11-1 is useful for comparing the properties of three categories of synchronization objects available to multi-threaded applications.

Table 11-1. Properties of Synchronization Objects

Characteristics	Operating System Synchronization Objects	Light Weight User Synchronization	Synchronization Object Based on MONITOR/MWAIT
Cycles to acquire and release (if there is a contention).	Thousands/Tens of thousands cycles.	Hundreds of cycles.	Hundreds of cycles.
Power Consumption	Saves power by halting the core or logical processor if idle.	Some power saving if using PAUSE.	Saves more power than PAUSE.
Scheduling and Context Switching.	Returns to the OS scheduler if contention exists (can be tuned with earlier spin loop count).	Does not return to OS scheduler voluntarily.	Does not return to OS scheduler voluntarily.
Ring level	Ring 0	Ring 3	Ring 0
Miscellaneous	Some objects provide intra-process synchronization and some are for inter-process communication.	Must lock accesses to synchronization variable if several threads may write to it simultaneously. Otherwise can write without locks.	Same as light weight. Can be used only on systems supporting MONITOR/MWAIT
Recommended use conditions	Number of active threads is greater than number of cores Waiting thousands of cycles for a signal Synchronization among processes	Number of active threads is less than or equal to number of cores Infrequent contention Need inter process synchronization	Same as light weight objects MONITOR/MWAIT available

11.4.2 SYNCHRONIZATION FOR SHORT PERIODS

The frequency and duration that a thread needs to synchronize with other threads depends application characteristics. When a synchronization loop needs very fast response, applications may use a spin-wait loop.

A spin-wait loop is typically used when one thread needs to wait a short amount of time for another thread to reach a point of synchronization. A spin-wait loop consists of a loop that compares a synchronization variable with some predefined value. See Example 11-4(a).

On a modern microprocessor with a superscalar speculative execution engine, a loop like this results in the issue of multiple simultaneous read requests from the spinning thread. These requests usually execute out-of-order with each read request being allocated a buffer resource. On detection of a write by a worker thread to a load that is in progress, the processor must guarantee no violations of memory order occur. The necessity of maintaining the order of outstanding memory operations inevitably costs the processor a severe penalty that impacts all threads.

This penalty occurs on the the Intel Core Solo and Intel Core Duo processors. However, the penalty on these processors is small compared with penalties suffered on the Intel Xeon processors. There the performance penalty for exiting the loop is about 25 times more severe.

On a processor supporting Intel HT Technology, spin-wait loops can consume a significant portion of the execution bandwidth of the processor. One logical processor executing a spin-wait loop can severely impact the performance of the other logical processor.

Example 11-4. Spin-wait Loop and PAUSE Instructions

(a) An un-optimized spin-wait loop experiences performance penalty when exiting the loop. It consumes execution resources without contributing computational work.

```
do {
    // This loop can run faster than the speed of memory access,
    // other worker threads cannot finish modifying sync_var until
    // outstanding loads from the spinning loops are resolved.
} while( sync_var != constant_value);
```

(b) Inserting the PAUSE instruction in a fast spin-wait loop prevents performance-penalty to the spinning thread and the worker thread

```
do {
_asm   pause
    // Ensure this loop is de-pipelined, i.e. preventing more than one
    // load request to sync_var to be outstanding,
    // avoiding performance penalty when the worker thread updates
    // sync_var and the spinning thread exiting the loop.

}
while( sync_var != constant_value);
```

(c) A spin-wait loop using a “test, test-and-set” technique to determine the availability of the synchronization variable. This technique is recommended when writing spin-wait loops to run on Intel 64 and IA-32 architecture processors.

Example 11-4. Spin-wait Loop and PAUSE Instructions (Contd.)

```
Spin_Lock:
    CMP lockvar, 0 ;            // Check if lock is free.
    JE  Get_lock
        PAUSE;                  // Short delay.
        JMP Spin_Lock;
Get_Lock:
        MOV EAX, 1;
        XCHG EAX, lockvar;      // Try to get lock.
        CMP EAX, 0;             // Test if successful.
        JNE Spin_Lock;
Critical_Section:
        <critical section code>
        MOV lockvar, 0;         // Release lock.
```

User/Source Coding Rule 14. (M impact, H generality) *Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.*

The penalty of exiting from a spin-wait loop can be avoided by inserting a PAUSE instruction in the loop. In spite of the name, the PAUSE instruction improves performance by introducing a slight delay in the loop and effectively causing the memory read requests to be issued at a rate that allows immediate detection of any store to the synchronization variable. This prevents the occurrence of a long delay due to memory order violation.

One example of inserting the PAUSE instruction in a simplified spin-wait loop is shown in Example 11-4(b). The PAUSE instruction is compatible with all Intel 64 and IA-32 processors. On IA-32 processors prior to Intel NetBurst microarchitecture, the PAUSE instruction is essentially a NOP instruction. Inserting the PAUSE instruction has the added benefit of significantly reducing the power consumed during the spin-wait because fewer system resources are used.

11.4.3 OPTIMIZATION WITH SPIN-LOCKS

Spin-locks are typically used when several threads needs to modify a synchronization variable and the synchronization variable must be protected by a lock to prevent unintentional overwrites. When the lock is released, however, several threads may compete to acquire it at once. Such thread contention significantly reduces performance scaling with respect to frequency, number of discrete processors, and Intel HT Technology.

To reduce the performance penalty, one approach is to reduce the likelihood of many threads competing to acquire the same lock. Apply a software pipelining technique to handle data that must be shared between multiple threads.

Instead of allowing multiple threads to compete for a given lock, no more than two threads should have write access to a given lock. If an application must use spin-locks, include the PAUSE instruction in the wait loop. Example 11-4(c) shows an example of the “test, test-and-set” technique for determining the availability of the lock in a spin-wait loop.

User/Source Coding Rule 15. (M impact, L generality) *Replace a spin lock that may be acquired by multiple threads with pipelined locks so no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to use a lock.*

11.4.4 SYNCHRONIZATION FOR LONGER PERIODS

When using a spin-wait loop not expected to be released quickly, an application should follow these guidelines:

- Keep the duration of the spin-wait loop to a minimum number of repetitions.

- Applications should use an OS service to block the waiting thread; this can release the processor so that other runnable threads can make use of the processor or available execution resources.

On processors supporting Intel HT Technology, operating systems should use the HLT instruction if one logical processor is active and the other is not. HLT will allow an idle logical processor to transition to a halted state; this allows the active logical processor to use all the hardware resources in the physical package. An operating system that does not use this technique must still execute instructions on the idle logical processor that repeatedly check for work. This "idle loop" consumes execution resources that could otherwise be used to make progress on the other active logical processor.

If an application thread must remain idle for a long time, the application should use a thread blocking API or other method to release the idle processor. The techniques discussed here apply to traditional MP system, but they have an even higher impact on processors that support Intel HT Technology.

Typically, an operating system provides timing services, for example Sleep(*dwMilliseconds*)[1]; such variables can be used to prevent frequent checking of a synchronization variable.

Another technique to synchronize between worker threads and a control loop is to use a thread-blocking API provided by the OS. Using a thread-blocking API allows the control thread to use less processor cycles for spinning and waiting. This gives the OS more time quanta to schedule the worker threads on available processors. Furthermore, using a thread-blocking API also benefits from the system idle loop optimization that OS implements using the HLT instruction.

User/Source Coding Rule 16. (H impact, M generality) *Use a thread-blocking API in a long idle loop to free up the processor.*

Using a spin-wait loop in a traditional MP system may be less of an issue when the number of runnable threads is less than the number of processors in the system. If the number of threads in an application is expected to be greater than the number of processors (either one processor or multiple processors), use a thread-blocking API to free up processor resources. A multithreaded application adopting one control thread to synchronize multiple worker threads may consider limiting worker threads to the number of processors in a system and use thread-blocking APIs in the control thread.

11.4.4.1 Avoid Coding Pitfalls in Thread Synchronization

Synchronization between multiple threads must be designed and implemented with care to achieve good performance scaling with respect to the number of discrete processors and the number of logical processor per physical processor. No single technique is a universal solution for every synchronization situation.

The pseudo-code example in Example 11-5(a) illustrates a polling loop implementation of a control thread. If there is only one runnable worker thread, an attempt to call a timing service API, such as Sleep(0), may be ineffective in minimizing the cost of thread synchronization. Because the control thread still behaves like a fast spinning loop, the only runnable worker thread must share execution resources with the spin-wait loop if both are running on the same physical processor that supports HT Technology. If there are more than one runnable worker threads, then calling a thread blocking API, such as Sleep(0), could still release the processor running the spin-wait loop, allowing the processor to be used by another worker thread instead of the spinning loop.

A control thread waiting for the completion of worker threads can usually implement thread synchronization using a thread-blocking API or a timing service, if the worker threads require significant time to complete. Example 11-5(b) shows an example that reduces the overhead of the control thread in its thread synchronization.

---

1. The Sleep() API is not thread-blocking, because it does not guarantee the processor will be released. Example 11-5(a) shows an example of using Sleep(0), which does not always realize the processor to another thread.

In general, OS function calls should be used with care when synchronizing threads. When using OS-supported thread

Example 11-5. Coding Pitfall using Spin Wait Loop

```
(a) A spin-wait loop attempts to release the processor incorrectly. It experiences a performance penalty if the
only worker thread and the control thread runs on the same physical processor package.
// Only one worker thread is running,
//  the control loop waits for the worker thread to complete.

ResumeWorkThread(thread_handle);
While (!task_not_done ) {
   Sleep(0)   // Returns immediately back to spin loop.
 …
}
(b) A polling loop frees up the processor correctly.

// Let a worker thread run and wait for completion.
ResumeWorkThread(thread_handle);
While (!task_not_done ) {
  Sleep(FIVE_MILISEC)

//  This processor is released for some duration, the processor
//  can be used by other threads.
…
}
```

synchronization objects (critical section, mutex, or semaphore), preference should be given to the OS service that has the least synchronization overhead, such as a critical section.

11.4.5 PREVENT SHARING OF MODIFIED DATA AND FALSE-SHARING

Depending on the cache topology relative to processor/core topology and the specific underlying microarchitecture, sharing of modified data can incur some degree of performance penalty when a software thread running on one core tries to read or write data that is currently present in modified state in the local cache of another core. This will cause eviction of the modified cache line back into memory and reading it into the first-level cache of the other core. The latency of such cache line transfer is much higher than using data in the immediate first level cache or second level cache.

False sharing applies to data used by one thread that happens to reside on the same cache line as different data used by another thread. These situations can also incur a performance delay depending on the topology of the logical processors/cores in the platform.

False sharing can experience a performance penalty when the threads are running on logical processors reside on different physical processors or processor cores. For processors that support HT Technology, false-sharing incurs a performance penalty when two threads run on different cores, different physical processors, or on two logical processors in the physical processor package. In the first two cases, the performance penalty is due to cache evictions to maintain cache coherency. In the latter case, performance penalty is due to memory order machine clear conditions.

A generic approach for multi-threaded software to prevent incurring false-sharing penalty is to allocate separate critical data or locks with alignment granularity according to a “false-sharing threshold” size. The following steps will allow software to determine the “false-sharing threshold” across Intel processors:

1. If the processor supports CLFLUSH instruction, i.e. CPUID.01H:EDX.CLFLUSH[bit 19] =1:

   — Use the CLFLUSH line size, i.e. the integer value of CPUID.01H:EBX[15:8], as the "false-sharing threshold".
2. If CLFLUSH line size is unavailable, use CPUID leaf 4 as described below:
   — Determine the "false-sharing threshold" by evaluating the largest system coherency line size among valid cache types that are reported via the sub-leaves of CPUID leaf 4. For each sub-leaf n, its associated system coherency line size is (CPUID.(EAX=4, ECX=n):EBX[11:0] + 1).
3. If neither CLFLUSH line size is available, nor CPUID leaf 4 is available, then software may choose the "false-sharing threshold" from one of the following:
   — Query the descriptor tables of CPUID leaf 2 and choose from available descriptor entries.
   — A Family/Model-specific mechanism available in the platform or a Family/Model-specific known value.
   — Default to a safe value 64 bytes.

User/Source Coding Rule 17. (H impact, M generality) *Beware of false sharing within a cache line or within a sector. Allocate critical data or locks separately using alignment granularity not smaller than the "false-sharing threshold".*

When a common block of parameters is passed from a parent thread to several worker threads, it is desirable for each work thread to create a private copy (each copy aligned to multiples of the *"false-sharing threshold")* of frequently accessed data in the parameter block.

11.4.6 PLACEMENT OF SHARED SYNCHRONIZATION VARIABLE

On processors based on Intel NetBurst microarchitecture, bus reads typically fetch 128 bytes into a cache, the optimal spacing to minimize eviction of cached data is 128 bytes. To prevent false-sharing, synchronization variables and system objects (such as a critical section) should be allocated to reside alone in a 128-byte region and aligned to a 128-byte boundary.

Example 11-6 shows a way to minimize the bus traffic required to maintain cache coherency in MP systems. This technique is also applicable to MP systems using processors with or without Intel HT Technology.

Example 11-6. Placement of Synchronization and Regular Variables

```
int regVar;
int padding[32];
int SynVar[32*NUM_SYNC_VARS];
int AnotherVar;
```

On processors based on Intel Core microarchitecture; a synchronization variable should be placed alone and in separate cache line to avoid false-sharing. Software must not allow a synchronization variable to span across page boundary.

User/Source Coding Rule 18. (M impact, ML generality) *Place each synchronization variable alone, separated by 128 bytes or in a separate cache line.*

User/Source Coding Rule 19. (H impact, L generality) *Do not place any spin lock variable to span a cache line boundary.*

At the code level, false sharing is a special concern in the following cases:

- Global data variables and static data variables that are placed in the same cache line and are written by different threads.
- Objects allocated dynamically by different threads may share cache lines. Make sure that the variables used locally by one thread are allocated in a manner to prevent sharing the cache line with other threads.

Another technique to enforce alignment of synchronization variables and to avoid a cacheline being shared is to use compiler directives when declaring data structures. See Example 11-7.

Example 11-7. Declaring Synchronization Variables without Sharing a Cache Line

```
__declspec(align(64)) unsigned __int64 sum;
struct sync_struct {…};
__declspec(align(64)) struct sync_struct sync_var;
```

Other techniques that prevent false-sharing include:

- Organize variables of different types in data structures (because the layout that compilers give to data variables might be different than their placement in the source code).
- When each thread needs to use its own copy of a set of variables, declare the variables with:
  - Directive threadprivate, when using OpenMP.
  - Modifier __declspec (thread), when using Microsoft compiler.
- In managed environments that provide automatic object allocation, the object allocators and garbage collectors are responsible for layout of the objects in memory so that false sharing through two objects does not happen.
- Provide classes so only one thread writes to each object field and close object fields, to avoid false sharing.

One should not equate the recommendations discussed in this section as favoring a sparsely populated data layout. The data-layout recommendations should be adopted when necessary and avoid unnecessary bloat in the size of the work set.

11.5 SYSTEM BUS OPTIMIZATION

The system bus services requests from bus agents (e.g. logical processors) to fetch data or code from the memory sub-system. The performance impact due data traffic fetched from memory depends on the characteristics of the workload, and the degree of software optimization on memory access, locality enhancements implemented in the software code. Several techniques to characterize memory traffic of a workload is discussed in Appendix A, "Application Performance Tools". Optimization guidelines on locality enhancement is also discussed in Section 3.6.10 and Section 9.5.11

The techniques described in Chapter 3, "General Optimization Guidelines" and Chapter 9, "Optimizing Cache Usage" benefit application performance in a platform where the bus system is servicing a single-threaded environment. In a multi-threaded environment, the bus system typically services many more logical processors, each of which can issue bus requests independently. Thus, techniques on locality enhancements, conserving bus bandwidth, reducing large-stride-cache-miss-delay can have strong impact on processor scaling performance.

11.5.1 CONSERVE BUS BANDWIDTH

In a multithreading environment, bus bandwidth may be shared by memory traffic originated from multiple bus agents (These agents can be several logical processors and/or several processor cores). Preserving the bus bandwidth can improve processor scaling performance. Also, effective bus bandwidth typically will decrease if there are significant large-stride cache-misses. Reducing the amount of large-stride cache misses (or reducing DTLB misses) will alleviate the problem of bandwidth reduction due to large-stride cache misses.

One way for conserving available bus command bandwidth is to improve the locality of code and data. Improving the locality of data reduces the number of cache line evictions and requests to fetch data. This technique also reduces the number of instruction fetches from system memory.

User/Source Coding Rule 20. (M impact, H generality) *Improve data and code locality to conserve bus command bandwidth.*

Using a compiler that supports profiler-guided optimization can improve code locality by keeping frequently used code paths in the cache. This reduces instruction fetches. Loop blocking can also improve the data locality. Other locality enhancement techniques can also be applied in a multithreading environment to conserve bus bandwidth (see Section 9.5).

Because the system bus is shared between many bus agents (logical processors or processor cores), software tuning should recognize symptoms of the bus approaching saturation. One useful technique is to examine the queue depth of bus read traffic. When the bus queue depth is high, locality enhancement to improve cache utilization will benefit performance more than other techniques, such as inserting more software prefetches or masking memory latency with overlapping bus reads. An approximate working guideline for software to operate below bus saturation is to check if bus read queue depth is significantly below 5.

Some MP and workstation platforms may have a chipset that provides two system buses, with each bus servicing one or more physical processors. The guidelines for conserving bus bandwidth described above also applies to each bus domain.

11.5.2 UNDERSTAND THE BUS AND CACHE INTERACTIONS

Be careful when parallelizing code sections with data sets that results in the total working set exceeding the second-level cache and /or consumed bandwidth exceeding the capacity of the bus. On an Intel Core Duo processor, if only one thread is using the second-level cache and / or bus, then it is expected to get the maximum benefit of the cache and bus systems because the other core does not interfere with the progress of the first thread. However, if two threads use the second-level cache concurrently, there may be performance degradation if one of the following conditions is true:

- Their combined working set is greater than the second-level cache size.
- Their combined bus usage is greater than the capacity of the bus.
- They both have extensive access to the same set in the second-level cache, and at least one of the threads writes to this cache line.

To avoid these pitfalls, multithreading software should try to investigate parallelism schemes in which only one of the threads access the second-level cache at a time, or where the second-level cache and the bus usage does not exceed their limits.

11.5.3 AVOID EXCESSIVE SOFTWARE PREFETCHES

Intel Xeon Processors have an automatic hardware prefetcher. It can bring data and instructions into the unified second-level cache based on prior reference patterns. In most situations, the hardware prefetcher is likely to reduce system memory latency without explicit intervention from software prefetches. It is also preferable to adjust data access patterns in the code to take advantage of the characteristics of the automatic hardware prefetcher to improve locality or mask memory latency. Processors based on Intel Core microarchitecture also provides several advanced hardware prefetching mechanisms. Data access patterns that can take advantage of earlier generations of hardware prefetch mechanism generally can take advantage of more recent hardware prefetch implementations.

Using software prefetch instructions excessively or indiscriminately will inevitably cause performance penalties. This is because excessively or indiscriminately using software prefetch instructions wastes the command and data bandwidth of the system bus.

Using software prefetches delays the hardware prefetcher from starting to fetch data needed by the processor core. It also consumes critical execution resources and can result in stalled execution. In some cases, it may be fruitful to evaluate the reduction or removal of software prefetches to migrate towards more effective use of hardware prefetch mechanisms. The guidelines for using software prefetch instructions are described in Chapter 3. The techniques for using automatic hardware prefetcher is discussed in Chapter 9.

User/Source Coding Rule 21. (M impact, L generality) *Avoid excessive use of software prefetch instructions and allow automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.*

11.5.4 IMPROVE EFFECTIVE LATENCY OF CACHE MISSES

System memory access latency due to cache misses is affected by bus traffic. This is because bus read requests must be arbitrated along with other requests for bus transactions. Reducing the number of outstanding bus transactions helps improve effective memory access latency.

One technique to improve effective latency of memory read transactions is to use multiple overlapping bus reads to reduce the latency of sparse reads. In situations where there is little locality of data or when memory reads need to be arbitrated with other bus transactions, the effective latency of scattered memory reads can be improved by issuing multiple memory reads back-to-back to overlap multiple outstanding memory read transactions. The average latency of back-to-back bus reads is likely to be lower than the average latency of scattered reads interspersed with other bus transactions. This is because only the first memory read needs to wait for the full delay of a cache miss.

User/Source Coding Rule 22. (M impact, M generality) *Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies.*

Another technique to reduce effective memory latency is possible if one can adjust the data access pattern so the access strides causing successive cache misses in the last-level cache is predominantly less than the trigger threshold distance of the automatic hardware prefetcher. See Section 9.5.3

User/Source Coding Rule 23. (M impact, M generality) *Consider adjusting the sequencing of memory references so the distribution of distances of successive cache misses of the last level cache peaks towards 64 bytes.*

11.5.5 USE FULL WRITE TRANSACTIONS TO ACHIEVE HIGHER DATA RATE

Write transactions across the bus can result in write to physical memory either using the full line size of 64 bytes or less than the full line size. The latter is referred to as a partial write. Typically, writes to writeback (WB) memory addresses are full-size and writes to write-combine (WC) or uncacheable (UC) type memory addresses result in partial writes. Both cached WB store operations and WC store operations utilize a set of six WC buffers (64 bytes wide) to manage the traffic of write transactions. When competing traffic closes a WC buffer before all writes to the buffer are finished, this results in a series of 8-byte partial bus transactions rather than a single 64-byte write transaction.

User/Source Coding Rule 24. (M impact, M generality) *Use full write transactions to achieve higher data throughput.*

Frequently, multiple partial writes to WC memory can be combined into full-sized writes using a software write-combining technique to separate WC store operations from competing with WB store traffic. To implement software write-combining, uncacheable writes to memory with the WC attribute are written to a small, temporary buffer (WB type) that fits in the first level data cache. When the temporary buffer is full, the application copies the content of the temporary buffer to the final WC destination.

When partial-writes are transacted on the bus, the effective data rate to system memory is reduced to only 1/8 of the system bus bandwidth.

11.6 MEMORY OPTIMIZATION

Efficient operation of caches is a critical aspect of memory optimization. Efficient operation of caches needs to address the following:

- Cache blocking.

- Shared memory optimization.
- Eliminating 64-KByte aliased data accesses.
- Preventing excessive evictions in first-level cache.

11.6.1 CACHE BLOCKING TECHNIQUE

Loop blocking is useful for reducing cache misses and improving memory access performance. The selection of a suitable block size is critical when applying the loop blocking technique. Loop blocking is applicable to single-threaded applications as well as to multithreaded applications running on processors with or without Intel HT Technology. The technique transforms the memory access pattern into blocks that efficiently fit in the target cache size.

When targeting Intel processors supporting HT Technology, the loop blocking technique for a unified cache can select a block size that is no more than one half of the target cache size, if there are two logical processors sharing that cache. The upper limit of the block size for loop blocking should be determined by dividing the target cache size by the number of logical processors available in a physical processor package. Typically, some cache lines are needed to access data that are not part of the source or destination buffers used in cache blocking, so the block size can be chosen between one quarter to one half of the target cache (see Chapter 3, "General Optimization Guidelines").

Software can use the deterministic cache parameter leaf of CPUID to discover which subset of logical processors are sharing a given cache (see Chapter 9, "Optimizing Cache Usage"). Therefore, guideline above can be extended to allow all the logical processors serviced by a given cache to use the cache simultaneously, by placing an upper limit of the block size as the total size of the cache divided by the number of logical processors serviced by that cache. This technique can also be applied to single-threaded applications that will be used as part of a multitasking workload.

User/Source Coding Rule 25. (H impact, H generality) *Use cache blocking to improve locality of data access. Target one quarter to one half of the cache size when targeting Intel processors supporting HT Technology or target a block size that allow all the logical processors serviced by a cache to share that cache simultaneously.*

11.6.2 SHARED-MEMORY OPTIMIZATION

Maintaining cache coherency between discrete processors frequently involves moving data across a bus that operates at a clock rate substantially slower that the processor frequency.

11.6.2.1 Minimize Sharing of Data between Physical Processors

When two threads are executing on two physical processors and sharing data, reading from or writing to shared data usually involves several bus transactions (including snooping, request for ownership changes, and sometimes fetching data across the bus). A thread accessing a large amount of shared memory is likely to have poor processor-scaling performance.

User/Source Coding Rule 26. (H impact, M generality) *Minimize the sharing of data between threads that execute on different bus agents sharing a common bus. The situation of a platform consisting of multiple bus domains should also minimize data sharing across bus domains.*

One technique to minimize sharing of data is to copy data to local stack variables if it is to be accessed repeatedly over an extended period. If necessary, results from multiple threads can be combined later by writing them back to a shared memory location. This approach can also minimize time spent to synchronize access to shared data.

11.6.2.2 Batched Producer-Consumer Model

The key benefit of a threaded producer-consumer design, shown in Figure 11-5, is to minimize bus traffic while sharing data between the producer and the consumer using a shared second-level cache. On an Intel Core Duo processor and when the work buffers are small enough to fit within the first-level cache, re-ordering of producer and

consumer tasks are necessary to achieve optimal performance. This is because fetching data from L2 to L1 is much faster than having a cache line in one core invalidated and fetched from the bus.

Figure 11-5 illustrates a batched producer-consumer model that can be used to overcome the drawback of using small work buffers in a standard producer-consumer model. In a batched producer-consumer model, each scheduling quanta batches two or more producer tasks, each producer working on a designated buffer. The number of tasks to batch is determined by the criteria that the total working set be greater than the first-level cache but smaller than the second-level cache.

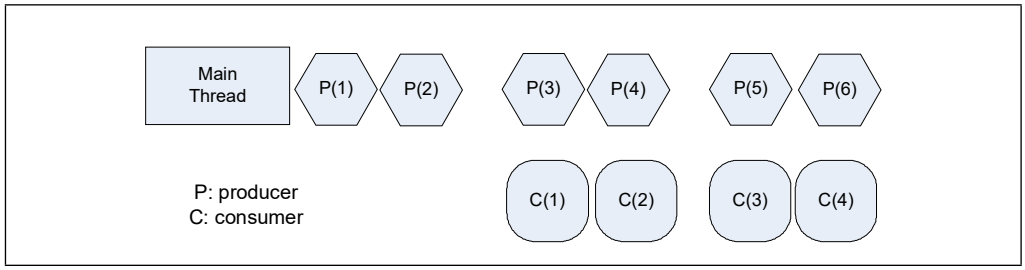


Figure 11-5. Batched Approach of Producer Consumer Model

Example 11-8 shows the batched implementation of the producer and consumer thread functions.

Example 11-8. Batched Implementation of the Producer Consumer Threads

```
void producer_thread()
{    int iter_num = workamount - batchsize;
     int mode1;
for (mode1=0; mode1 < batchsize; mode1++)
{          produce(buffs[mode1],count);  }

     while (iter_num--)
     {     Signal(&signal1,1);
           produce(buffs[mode1],count); // placeholder function
           WaitForSignal(&end1);
           mode1++;
           if (mode1 > batchsize)
                mode1 = 0;
     }
}

void consumer_thread()
{    int mode2 = 0;
     int iter_num = workamount - batchsize;
     while (iter_num--)
     {     WaitForSignal(&signal1);
           consume(buffs[mode2],count); // placeholder function
           Signal(&end1,1);
           mode2++;
           if (mode2 > batchsize)
                mode2 = 0;
```

Example 11-8. Batched Implementation of the Producer Consumer Threads (Contd.)

```
    }
    for (i=0;i<batchsize;i++)
    {     consume(buffs[mode2],count);
          mode2++;
          if (mode2 > batchsize)
               mode2 = 0;
    }
}
```

11.6.3 ELIMINATE 64-KBYTE ALIASED DATA ACCESSES

The 64-KByte aliasing condition is discussed in Chapter 3, "General Optimization Guidelines". Memory accesses that satisfy the 64-KByte aliasing condition can cause excessive evictions of the first-level data cache. Eliminating 64-KByte aliased data accesses originating from each thread helps improve frequency scaling in general. Furthermore, it enables the first-level data cache to perform efficiently when Intel HT Technology is fully utilized by software applications.

User/Source Coding Rule 27. (H impact, H generality) *Minimize data access patterns that are offset by multiples of 64 KBytes in each thread.*

Performance penalties associated with 64-KByte aliasing are applicable mainly to current processor implementations of Intel HT Technology or Intel NetBurst microarchitecture. The next section discusses memory optimization techniques that are applicable to multithreaded applications running on processors supporting Intel HT Technology.

11.7 FRONT END OPTIMIZATION

For dual-core processors where the second-level unified cache is shared by two processor cores (Intel Core Duo processor and processors based on Intel Core microarchitecture), multi-threaded software should consider the increase in code working set due to two threads fetching code from the unified cache as part of front end and cache optimization. For quad-core processors based on Intel Core microarchitecture, the considerations that applies to Intel Core 2 Duo processors also apply to quad-core processors.

11.7.1 AVOID EXCESSIVE LOOP UNROLLING

Unrolling loops can reduce the number of branches and improve the branch predictability of application code. Loop unrolling is discussed in detail in Chapter 3, "General Optimization Guidelines". Loop unrolling must be used judiciously. Be sure to consider the benefit of improved branch predictability and the cost of under-utilization of the loop stream detector (LSD).

User/Source Coding Rule 28. (M impact, L generality) *Avoid excessive loop unrolling to ensure the LSD is operating efficiently.*

11.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES

Modern OSes provide either API and/or data constructs (e.g. affinity masks) that allow applications to manage certain shared resources , e.g. logical processors, Non-Uniform Memory Access (NUMA) memory sub-systems.

Before multithreaded software considers using affinity APIs, it should consider the recommendations in Table 11-2.

Table 11-2. Design-Time Resource Management Choices

Runtime Environment	Thread Scheduling/Processor Affinity Consideration	Memory Affinity Consideration
A single-threaded application	• Support OS scheduler objectives on system response and throughput by letting OS scheduler manage scheduling. • OS provides facilities for end user to optimize runtime specific environment.	Not relevant; let OS do its job.
A multi-threaded application requiring: • Less than all processor resource in the system. • Shares system resources with other concurrent applications. • Other concurrent applications may have higher priority.	• Rely on OS default scheduler policy. • Hard-coded affinity-binding will likely harm system response and throughput; and/or in some cases hurting application performance.	• Rely on OS default scheduler policy. • Use API that could provide transparent NUMA benefit without managing NUMA explicitly.
A multi-threaded application requiring • Foreground and higher priority. • Uses less than all processor resource in the system. • Share system resource with other concurrent applications. • However, other concurrent applications have lower priority.	If application-customized thread binding policy is considered, • A cooperative approach with OS scheduler should be taken instead of hard-coded thread affinity binding policy. • Example: the use of **SetThreadIdealProcessor()** can provide a floating base to anchor a next-free-core binding policy for a locality-optimized application binding policy, and cooperate with default OS policy.	• Use API that could provide transparent NUMA benefit without managing NUMA explicitly. • Use performance event to diagnose non-local memory access issue if default OS policy cause performance issue.
A multithreaded application runs in foreground, • Requires all processor resource in the system. • Doesn't share system resource with concurrent applications • Multithreading.	• Application-customized thread binding policy can be more efficient than default OS policy. • Use performance event to help optimize locality and cache transfer opportunities. • A multithreaded application that employs its own explicit thread affinity-binding policy should deploy with some form of opt-in choice granted by the end-user or administrator. • Example, permission to deploy explicit thread affinity-binding policy can be activated after permission is granted after installation.	• Application-customized memory affinity binding policy can be more efficient than default OS policy. • Use performance event to diagnose non-local memory access issues related to either OS or custom policy

11.8.1 TOPOLOGY ENUMERATION OF SHARED RESOURCES

Whether multithreaded software ride on OS scheduling policy or need to use affinity APIs for customized resource management, understanding the topology of the shared platform resource is essential. The processor topology of logical processors (SMT), processor cores, and physical processors in the platform can enumerated using information provided by CPUID. This is discussed in Chapter 9, "Multiple-Processor Management" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. A white paper and reference code is also available from Intel.

11.8.2 NON-UNIFORM MEMORY ACCESS (NUMA)

Platforms using two or more Intel Xeon processors based on Nehalem microarchitecture support non-uniform memory access (NUMA) topology because each physical processor provides its own local memory controller. NUMA offers system memory bandwidth that can scale with the number of physical processors. System memory latency will exhibit asymmetric behavior depending on the memory transaction occurring locally in the same socket or remotely from another socket. Additionally, OS-specific construct and/or implementation behavior may present additional complexity at the API level that the multi-threaded software may need to pay attention to memory allocation/initialization in a NUMA environment.

Generally, latency sensitive workload would favor memory traffic to stay local over remote. If multiple threads shares a buffer, the programmer will need to pay attention to OS-specific behavior of memory allocation/initialization on a NUMA system.

Bandwidth sensitive workloads will find it convenient to employ a data composition threading model and aggregates application threads executing in each socket to favor local traffic on a per-socket basis to achieve overall bandwidth scalable with the number of physical processors.

The OS construct that provides the programming interface to manage local/remote NUMA traffic is referred to as memory affinity. Because OS manages the mapping between physical address (populated by system RAM) to linear address (accessed by application software); and paging allows dynamic reassignment of a physical page to map to different linear address dynamically, proper use of memory affinity will require a great deal of OS-specific knowledge.

To simplify application programming, OS may implement certain APIs and physical/linear address mapping to take advantage of NUMA characteristics transparently in certain situations. One common technique is for OS to delay commit of physical memory page assignment until the first memory reference on that physical page is accessed in the linear address space by an application thread. This means that the allocation of a memory buffer in the linear address space by an application thread does not necessarily determine which socket will service local memory traffic when the memory allocation API returns to the program. However, the memory allocation API that supports this level of NUMA transparency varies across different OSes. For example, the portable C-language API "malloc" provides some degree of transparency on Linux*, whereas the API "VirtualAlloc" behave similarly on Windows*. Different OSes may also provide memory allocation APIs that require explicit NUMA information, so the mapping between linear address to local/remote memory traffic are fixed at allocation.

Example 11-9 shows an example that multi-threaded application could undertake the least amount of effort dealing with OS-specific APIs and to take advantage of NUMA hardware capability. This parallel approach to memory buffer initialization is conducive to having each worker thread keep memory traffic local on NUMA systems.

Example 11-9. Parallel Memory Initialization Technique Using OpenMP and NUMA

```
#ifdef _LINUX // Linux implements malloc to commit physical page at first touch/access
  buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
  buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
  buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
#endif
#ifdef windows
// Windows implements malloc to commit physical page at allocation, so use VirtualAlloc
  buf1 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
  buf2 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
  buf3 = (char *) VirtualAlloc(NULL, DIM*(sizeof (double))+1024, fAllocType, fProtect);
#endif

  a = (double *) buf1;
  b = (double *) buf2;
  c = (double *) buf3;
#pragma omp parallel
{ // use OpenMP threads to execute each iteration of the loop
// number of OpenMP threads can be specified by default or via environment variable
#pragma omp for private(num)
// each loop iteration is dispatched to execute in different OpenMP threads using private iterator
  for(num=0;num<len;num++)
  {// each thread perform first-touches to its own subset of memory address, physical pages
//    mapped to the local memory controller of the respective threads
        a[num]=10.;
        b[num]=10.;
        c[num]=10.;
  }
}
```

Note that the example shown in Example 11-9 implies that the memory buffers will be freed after the worker threads created by OpenMP have ended. This situation avoids a potential issue of repeated use of malloc/free across different application threads. Because if the local memory that was initialized by one thread and subsequently got freed up by another thread, the OS may have difficulty in tracking/re-allocating memory pools in linear address space relative to NUMA topology. In Linux, another API, “numa_local_alloc” may be used.

11.9 OPTIMIZATION OF OTHER SHARED RESOURCES

Resource optimization in multithreaded application depends on the cache topology and execution resources associated within the hierarchy of processor topology. Processor topology and an algorithm for software to identify the processor topology are discussed in Chapter 9, "Multiple-Processor Management" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

In platforms with shared buses, the bus system is shared by multiple agents at the SMT level and at the processor core level of the processor topology. Thus multithreaded application design should start with an approach to manage the bus bandwidth available to multiple processor agents sharing the same bus link in an equitable manner. This can be done by improving the data locality of an individual application thread or allowing two threads to take advantage of a shared second-level cache (where such shared cache topology is available).

In general, optimizing the building blocks of a multithreaded application can start from an individual thread.

Tuning Suggestion 2. *Optimize single threaded code to maximize execution throughput first.*

Tuning Suggestion 3. *Employ efficient threading model, leverage available tools (such as Intel Threading Building Block, Intel Thread Checker, Intel Thread Profiler) to achieve optimal processor scaling with respect to the number of physical processors or processor cores.*

11.9.1 EXPANDED OPPORTUNITY FOR INTEL® HT OPTIMIZATION

The Intel® Hyper-Threading Technology (Intel® HT) implementation in Nehalem microarchitecture differs from previous generations of Intel HT implementations. It offers broader opportunity for multithreaded software to take advantage of Intel HT and achieve higher system throughput over a broader range of application problems. This section provides a few heuristic recommendations and illustrates some of these optimization opportunities.

Chapter 2, "Intel® 64 and IA-32 Architectures" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1covered some of the microarchitectural capability enhancements in Intel Hyper-Threading Technology. Many of these enhancements center around the basic needs of multi-threaded software in terms of sharing common hardware resources that may be used by more than one thread context.

Different software algorithms and workload characteristics may produce different performance characteristics due to their demands on critical microarchitectural resources that may be shared amongst several logical processors. A brief comparison of the various microarchitectural subsystems that can play a significant role in software tuning for Intel HT is summarized in Table 11-3.

Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations

Microarchitectural Subsystem(s)	Nehalem Microarchitecture 06_1AH	NetBurst Microarchitecture 0F_02H, 0F_03H, 0F_04H, 0F_06H
Issue ports, execution units	Three issue ports (0, 1, 5) distributed to handle ALU, SIMD, and FP computations.	Unbalanced ports, fast ALU SIMD and FP sharing the same port (port 1).
Buffering	More entries in ROB, RS, fill buffers, etc., with moderate pipeline depths.	Less balance between buffer entries and pipeline depths.
Branch Prediction and Misaligned Memory Access	More robust speculative execution with immediate reclamation after misprediction; efficient handling of cache splits.	More microarchitectural hazards resulting in pipeline cleared for both threads.

Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations (Contd.)

Microarchitectural Subsystem(s)	Nehalem Microarchitecture 06_1AH	NetBurst Microarchitecture 0F_02H, 0F_03H, 0F_04H, 0F_06H
Cache Hierarchy	Larger and more efficient.	More microarchitectural hazards to work around.
Memory and Bandwidth	NUMA, three channels per socket to DDR3, up to 32GB/s per socket.	SMP, FSB, or dual FSB, up to 12.8 GB/s per FSB.

For compute bound workloads, the Intel HT opportunity in Intel NetBurst microarchitecture tends to favor thread contexts that executes with relatively high CPI (average cycles to retire consecutive instructions). At a hardware level, this is in part due to the issue port imbalance in the microarchitecture, as port 1 is shared by fast ALU, slow ALU (more heavy-duty integer operations), SIMD, and FP computations. At a software level, some of the cause for high CPI and may appear as benign catalyst for providing HT benefit may include: long latency instructions (port 1), some L2 hits, occasional branch mispredictions, etc. But the length of the pipeline in NetBurst microarchitecture often impose additional internal hardware constraints that limits software's ability to take advantage of Intel HT.

The microarchitectural enhancements listed in Table 11-3 are expected to provide broader software optimization opportunities for compute-bound workloads. Whereas contention in the same execution unit by two compute-bound threads might be a concern to choose a functional-decomposition threading model over data-composition threading. Nehalem microarchitecture will likely be more accommodating to support the programmer to choose the optimal threading decomposition models.

Memory intensive workloads can exhibit a wide range of performance characteristics, ranging from completely parallel memory traffic (saturating system memory bandwidth, as in the well-known example of Stream), memory traffic dominated by memory latency, or various mixtures of compute operations and memory traffic of either kind.

The Intel HT implementation in Intel NetBurst microarchitecture may provide benefit to some of the latter two types of workload characteristics. TheIntel HT capability in the Nehalem microarchitecture can broaden the operating envelop of the two latter types of workload characteristics to deliver higher system throughput, due to its support for non-uniform memory access (NUMA), more efficient link protocol, and system memory bandwidth that scales with the number of physical processors.

Some cache levels of the cache hierarchy may be shared by multiple logical processors. Using the cache hierarchy is an important means for software to improve the efficiency of memory traffic and avoid saturating the system memory bandwidth. Multi-threaded applications employing cache-blocking technique may wish to partition a target cache level to take advantage of Intel HT. Alternately two logical processors sharing the same L1 and L2, or logical processors sharing the L3 may wish to manage the shared resources according to their relative topological relationship. A white paper on processor topology enumeration and cache topology enumeration with companion reference code has been published.

CHAPTER 12
INTEL® OPTANE™ DC PERSISTENT MEMORY

The Intel® Xeon® scalable processor family based on the Cascade Lake product introduces support for Intel® Optane™ DC Persistent Memory Modules. These Intel Optane DC Persistent Memory Modules are larger in size compared to DRAM and are persistent, i.e., the content is maintained even when the system is powered down. However, latency is higher and bandwidth is lower than DRAM DIMMs.

12.1 MEMORY MODE AND APP-DIRECT MODE

Intel Optane DC Persistent Memory Module DIMMs can be used in two different modes.

12.1.1 MEMORY MODE

In memory mode, the memory is exposed as volatile memory. This is transparent to the operating system and applications. In particular, software can benefit, without modifications, from large memory capacity. The DRAM memory present in the system is being used as a memory-side cache. The intent behind this is for software to get the latency of the DRAM tier, while holding “in-memory” data that is the capacity of the Intel Optane DC Persistent Memory Module tier.

In memory mode, data on Intel Optane DC Persistent Memory Modules becomes inaccessible after a reboot. Since the media itself is non-volatile, this is implemented by encrypting the data with a key that is discarded during a power-cycle. The DRAM memory that is present on the socket is used as a directly-mapped cache for the Intel Optane DC Persistent Memory Modules. This implies that, in contrast to processor caches, there is no LRU policy for the cache. A cache line on Intel Optane DC Persistent Memory Modules will always evict the same cache line from DRAM. Operating systems can optimize for memory mode by using pages whose addresses do not conflict with pages that hold data that should not be evicted from the DRAM cache. For example, it is usually beneficial to always keep page tables in DRAM. The size of the working set greatly determines performance. If the working set of an application fits in DRAM, performance is not impacted as much by the latency and bandwidth of Intel Optane DC Persistent Memory Modules.

12.1.2 APP DIRECT MODE

In app direct mode, the memory is exposed as a device, which can be formatted with a file system. One option is to use the Intel Optane DC Persistent Memory Module as a very fast block device, with a file system, called “storage over app direct”. This has the advantage that applications that are I/O bound benefit from the Intel Optane DC Persistent Memory Module without modifications to the software. In other words, we are using app direct for getting a fast storage device, but not using it as persistent memory. In contrast with this usage, rewriting the application for usage as persistent memory has several key benefits. The key difference from usage as “storage over app direct” is that data can be accessed on a cache line granularity. To load or store data on the device, the processor load and store instructions are used and no OS interaction is needed, once a page is allocated to a process. Thus app direct mode implements persistent memory. The operating system does not access persistent memory as RAM, but rather through a file system mounted with a special flag called “dax”. This usage of “dax” is what differentiates “app direct mode” (mount with dax) from “storage over app direct mode” (mount without dax). Mounting with dax provides the following advantages.

Once the persistent memory is mapped to the virtual address space of the application, reads and writes can be done using load and store instructions, and this has the following advantages over storage over app direct:

- This completely avoids the system call.
- Instead of transferring a complete page (e.g. 4KB), only a cache line is transferred (64B).
- Only one copy of the data is needed as memory is not copied back and forth between persistent memory and DRAM.
- Access is synchronous.

Note that the memory the operating system and conventional OS memory reporting tools report only the DRAM that is present in the system, since persistent memory is accessible via a file system. The two pools of memory are distinct from one another, and one of the key differentiators of app direct mode is that software has full control over which data is located in DRAM vs. NVDIMM. For optimal performance, software can therefore place latency-sensitive data structures in DRAM, either by generating a copy or reconstructing it. Examples are index data structures, which are typically accessed randomly but can be reconstructed after a restart.

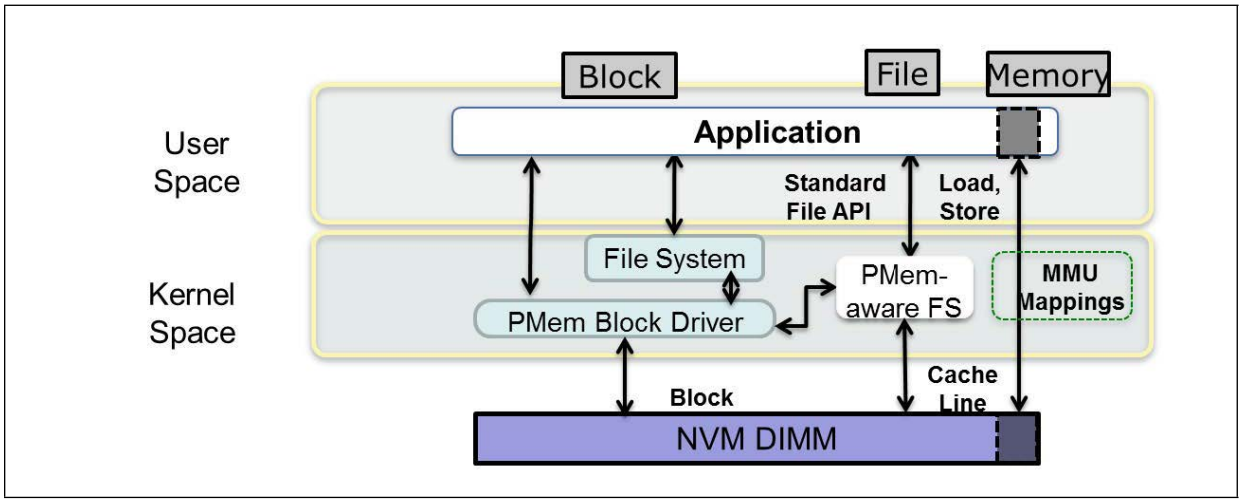


Figure 12-1. In App Direct Mode, Data on the Intel® Optane™ DC Persistent Memory Module is Accessed Directly with Loads and Stores

12.1.3 SELECTING A MODE

The software developer needs to consider various factors while determining which mode may be best suited for a given application and usage scenario.

- Is there a benefit from large memory capacity (larger than what is possible with DRAM on a given platform)? For example, an application may be paging heavily to disk, and may be able to page less with larger memory capacity. Other examples may be:
  - An application choosing a different algorithm when given larger memory capacity, which may result in better performance.
  - An application choosing to store and reuse intermediate results when given larger capacity.
- Is there a benefit from using persistence in the memory sub-system?
  - This may include faster start-up times (avoid loading data from disk to memory, and/or avoid re-building “in-memory” pointer-based structures like linked lists or trees on restart.
    - This may also include benefits from a faster path to durability.

    - For example, memory could be the final, durable destination for data instead of disk.
    - Applications that are bound by disk latency or bandwidth can benefit from using memory for durability.
- What is the sensitivity of the application to memory latency?
    - Intel Optane DC Persistent Memory Module latencies are higher than DRAM, typically around 3-4 times the latency of DRAM.
        - In the cases where an Intel Optane DC Persistent Memory Module is replacing memory, a lot depends on how predicable those accesses are, and also how sensitive those memory accesses are to latency.

To illustrate these cases, let's first consider the scenario where the application is reading a sequential array of numbers that is several GB in size from an Intel Optane DC Persistent Memory Module. In this case, since the accesses are spatially predictable, they are prefetchable by hardware and software prefetchers. As a result, the data can always be in the processor caches before the application requests the data, and the latency of the Intel Optane DC Persistent Memory Module is not seen by the application.

On the other hand, if the application was walking a linked list for example, it is not possible to identify the next node in the linked list without first reading the current node (this is called "pointer chasing"). In this case, the latency of the Intel Optane DC Persistent Memory Module is seen by the application.

Another important consideration mentioned above is the sensitivity of the application to memory latency. In some cases, the application is so the processor cores can do other useful work while waiting for memory references to the Intel Optane DC Persistent Memory Module to return; since useful work is being done, performance is often not significantly impacted.

In other cases, the cores are stalled while waiting for memory references from the Intel Optane DC Persistent Memory Module, which often impacts performance.

If the application as a whole is indeed sensitive to memory latency, an examination of which memory data structures are sensitive is warranted. A good use for the Intel Optane DC Persistent Memory Module is large capacity data structures that are not as sensitive to memory latency based on the considerations outlines above. Smaller data structures that are heavily accessed and/or are sensitive to memory latency are better suited to DRAM.

The chart below shows a pictorial flow based on the above considerations.

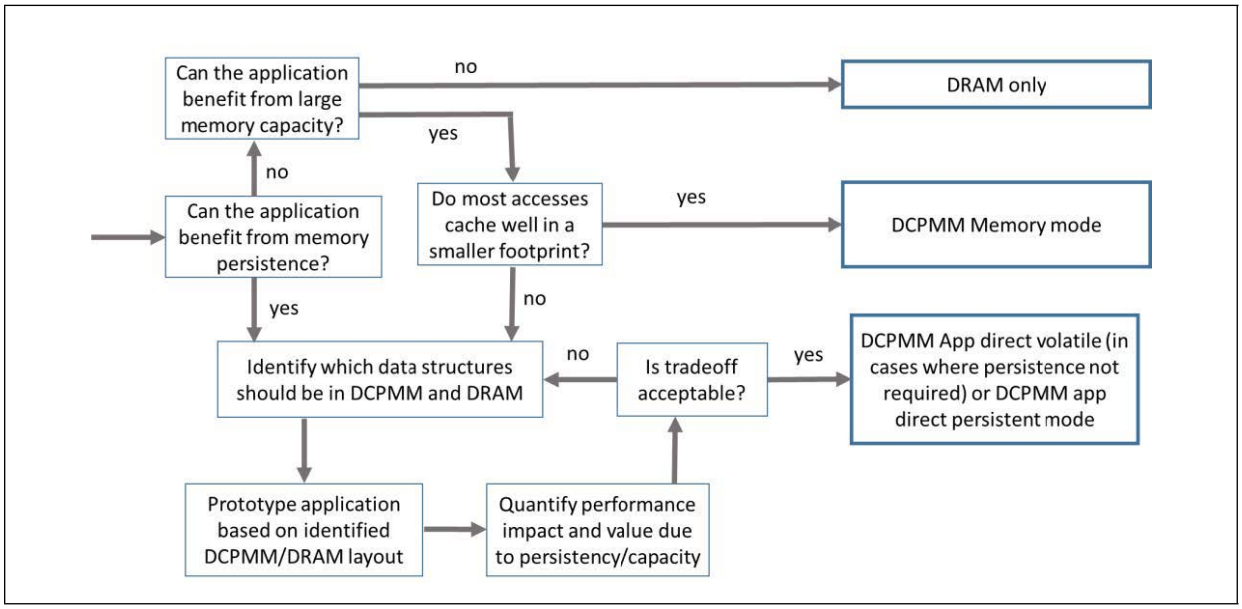


Figure 12-2. Decision Flow for Determining When to Use Intel® Optane™ DC Persistent Memory Module vs. DRAM

12.2 DEVICE CHARACTERISTICS OF INTEL® OPTANE™ DC PERSISTENT MEMORY MODULE

In the previous section, one of the considerations for software developers to select the Intel Optane DC Persistent Memory Module for a data structure was “performance sensitivity to memory latency”. In this section, we provide various considerations that determine this sensitivity; these include different device characteristics from DRAM, additional code changes required for new features like persistence in memory, etc.

12.2.1 INTEL® OPTANE™ DC PERSISTENT MEMORY MODULE LATENCY

Intel Optane DC Persistent Memory Module devices have different access characteristics from DRAM since they are made of a different material than DRAM. The table below summarizes read latencies for sequential and random accesses respectively.

Table 12-1. Latencies for Accessing Intel® Optane™ DC Persistent Memory Modules

Latency	Intel® Optane™ DC Persistent Memory Module	DRAM
Idle sequential read latency	~170ns	~75ns
Idle random read latency	~320ns	~80ns

In the case of DRAM, the difference between sequential and random latencies is limited to a few nanoseconds; this is due to sequential accesses resulting in greater hits in DRAM row buffers. However in the case of Intel Optane DC Persistent Memory Modules, not only do the latencies differ overall from DRAM, they also differ significantly between the sequential and random access cases.

The difference in access latency of Intel Optane DC Persistent Memory Modules from DRAM requires special consideration for software developers from a performance perspective.

See Chapter 9, “Optimizing Cache Usage” for general guidelines on optimizing processor cache usage.

In memory mode, it is expected that the DRAM cache would absorb most of the accesses, and the application would see DRAM-like latencies. Note that the latency to access Intel Optane DC Persistent Memory Modules in memory mode is ~30-40 ns higher than in app direct mode, due to the overhead of first looking up the DRAM cache. Performance in memory mode can be improved with traditional cache tiling and locality optimization techniques that keep the working set within the size of the DRAM cache.

Further, each Intel Optane DC Persistent Memory Module features some form of buffering at 256 Byte granularity, and this is one of the units at which we distinguish between sequential and random accesses. It is therefore beneficial to collocate data inside 256 Bytes and read them together to get sequential access latencies as opposed to random, a consideration for software data structure design.

12.2.2 READ VS. WRITE BANDWIDTH

The different access characteristics of Intel Optane DC Persistent Memory Modules include different read and write bandwidths compared to DRAM, as illustrated in the table below.

Table 12-2. Bandwidths per DIMM for Intel® Optane™ DC Persistent Memory Modules and DRAM

Per DIMM Bandwidths	Intel® Optane™ DC Persistent Memory Module	DRAM
Sequential read	~7.6 GB/s	~15 GB/s
Random read	~2.4 GB/s	~15 GB/s
Sequential write	~2.3 GB/s	~15 GB/s
Random write	~0.5 GB/s	~15 GB/s

From the above table, we can make the following observations.

1. Reads and writes are asymmetrical, and more specifically read bandwidths are higher than write bandwidths. This is an important consideration for software design, and should be factored in decisions. For example, a data structure with random writes and high write amplification would not be a good choice for Intel Optane DC Persistent Memory Modules.
2. Sequential and random access characteristics are also markedly different. This is again a consideration for choice of data structure design and placement in Intel Optane DC Persistent Memory Modules compared with DRAM, with emphasis on the locality within 256B granularity to get the benefits of sequential access placements.

It is important to note that bandwidth is a first class constraint in usage of these DIMMs and it is important to avoid operating at bandwidths close to the capability of the DIMM, as illustrated with the red circles in Figure 12-3.

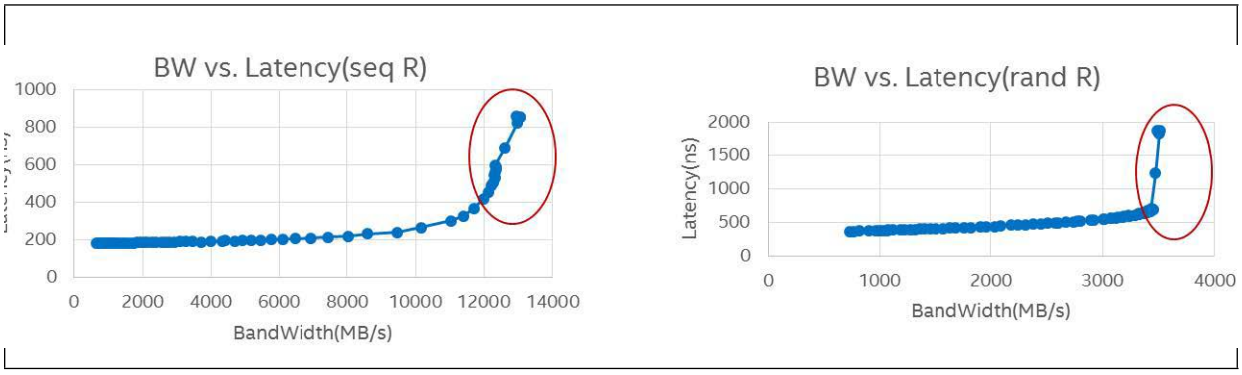


Figure 12-3. Loaded Latency Curves for One Intel® Optane™ DC Persistent Memory Module DIMM: Sequential Traffic (Left) and Random Traffic (Right)

When memory bandwidth is close to being saturated, the latencies tend to be very high and hurt application performance. The bandwidth demand is typically a function of the number of cores driving memory accesses, and the nature of the accesses, i.e., sequential vs. random access pattern as well as the read-write mix. On the other hand, the bandwidth capability of the platform is a function of the number of channels and DIMMs available.

It is therefore important to balance the read and write traffic with the capabilities of the system. Which is to say, the number of threads reading and writing to the Intel Optane DC Persistent Memory Module vs. the number of populated memory channels.

While writing to Intel Optane DC Persistent Memory Module, since bandwidth is more limited than for DRAM, it is recommended to use non-temporal stores over regular stores in cases when it is not expected that the data written to will be re-used in the near future, or while writing to very large buffers. (See Section 9.4.1.2 for details).

12.2.3 NUMBER OF THREADS FOR OPTIMAL BANDWIDTH

As noted earlier, the Intel Optane DC Persistent Memory Module DIMM is buffering and combining data at 256B granularity. This can have implications on the number of threads that are accessing memory on the Intel Optane DC Persistent Memory Module. If there are too many threads attempting to write to the memory on the Intel Optane DC Persistent Memory Module concurrently, the benefits of write combining and 256B locality are lost if spatial locality is lost due to writes intercepting from other threads.

As a result, even though each thread may write sequentially, the traffic begins to look random at the DIMM level, and therefore as the number of threads writing to the Intel Optane DC Persistent Memory Module crosses a threshold, one begins to observe random access bandwidths instead of sequential access bandwidths.

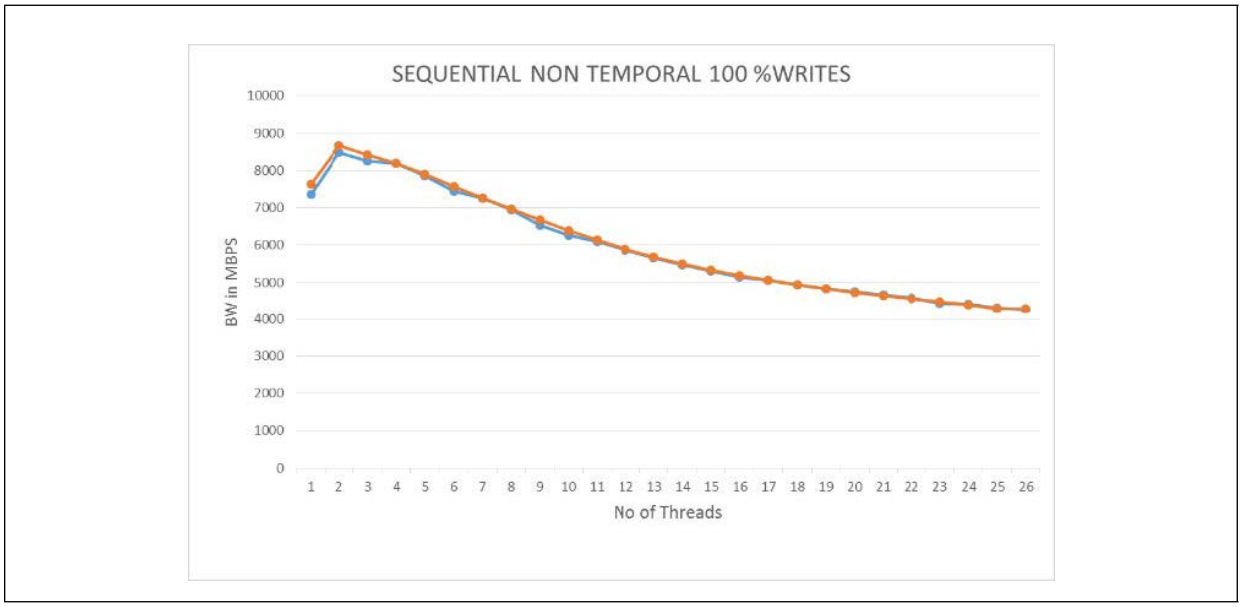


Figure 12-4. Number of Threads vs. Bandwidth[1]

NOTES:

1. As the number of threads increases, the bandwidth first increases and then decreases. This decrease is due to the fact that even though the accesses are sequential in nature, as we have more and more threads injecting their accesses into the memory subsystem, the “sequentiality” (especially at the mentioned 256B granularity) is lost when observed from the standpoint of a finite buffer for write combining.

Figure 12-5, Figure 12-6, and Figure 12-7 illustrate the differences in combining at 256B locality and how this is impacted by the number of threads that are injecting references to the Intel Optane DC Persistent Memory Module. It is important to keep this 256B locality in mind while selecting data structures, and the concurrency of accesses to the Intel Optane DC Persistent Memory Module.

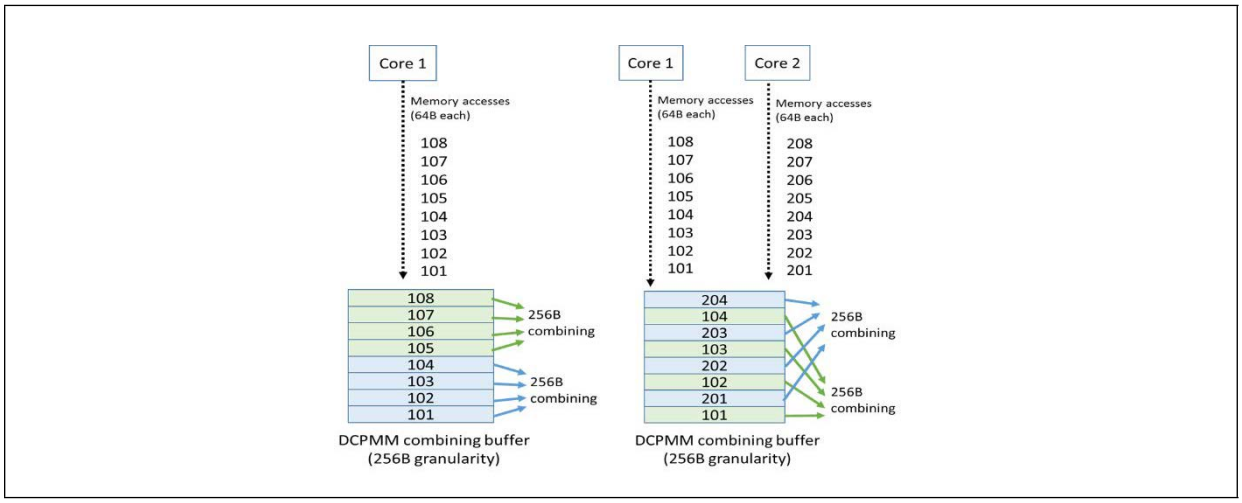


Figure 12-5. Combining with Two Cores[1]

NOTES:

1. 101, 102, etc. refer to 64B accesses from a core 1, and likewise for the other core. It can be seen that for two cores, and a sample buffer size of 8 (note that this is strictly an example size for illustration purposes), there is 100% combining within the buffer at 256B granularity. This makes the accesses 100% sequential from the memory system standpoint.

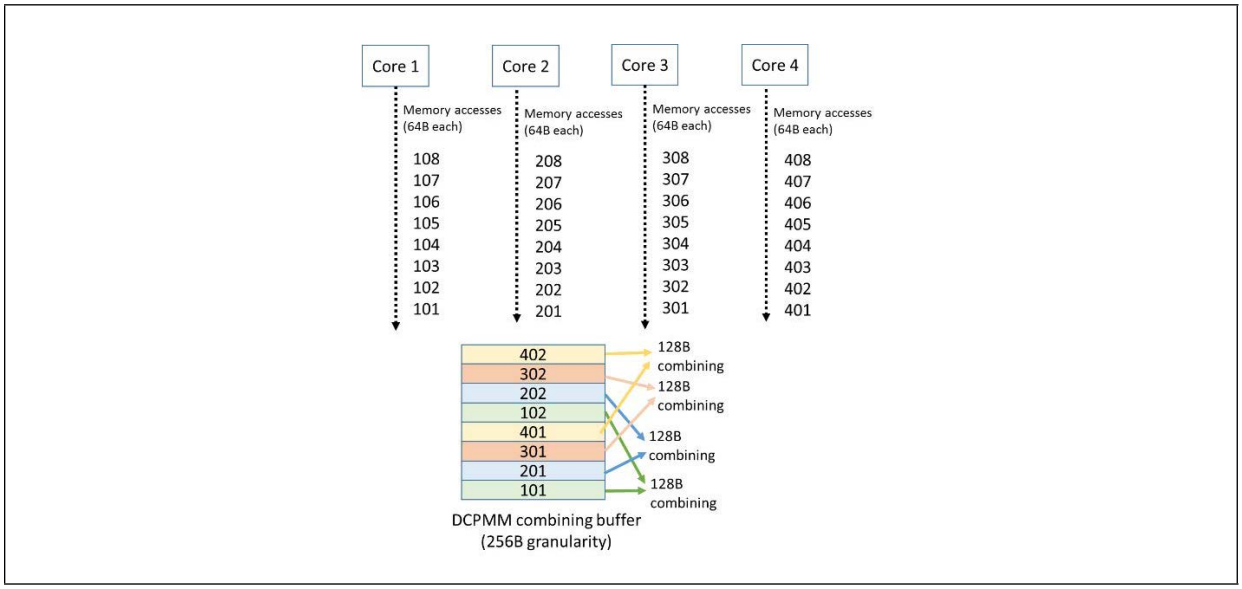


Figure 12-6. Combining with Four Cores[1]

NOTES:

1. 101, 102, etc refer to 64B accesses from core 1, and likewise for the other cores. It can be seen that for four cores, and a sample buffer size of 8, there is 50% combining within the buffer at 256B granularity (only 128B combining is possible as the buffer gets full and needs to be drained for forward progress). This makes the accesses 50% sequential from the memory system standpoint.

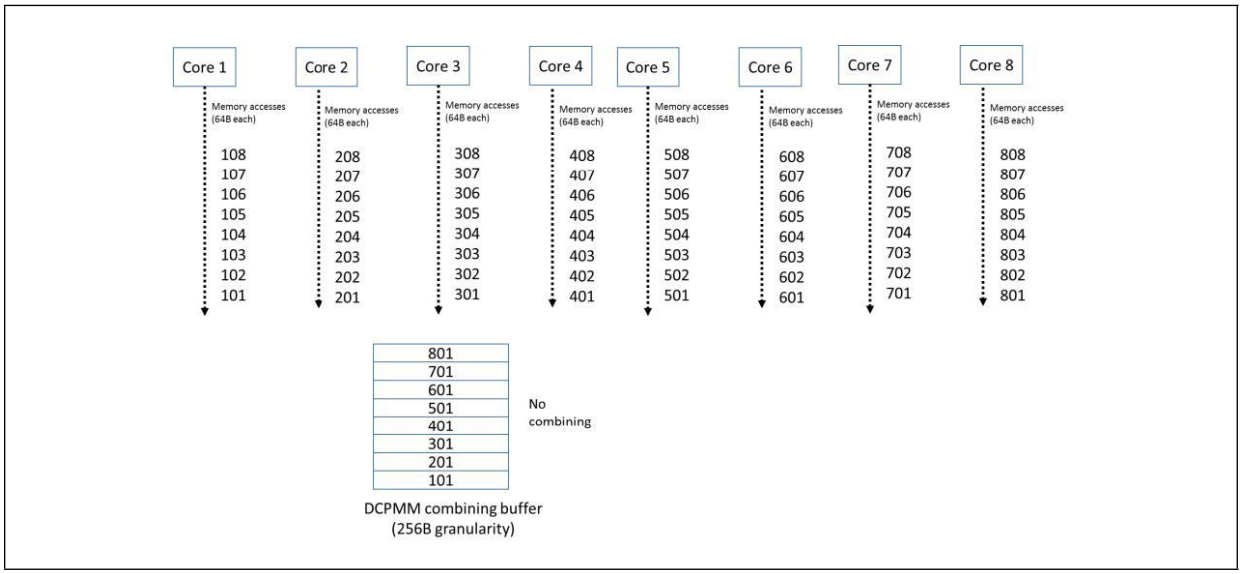


Figure 12-7. Combining with Eight Cores[1]

NOTES:

1. For eight cores, there is no combining possible and the sequential references seem random from a memory system standpoint.

12.3 PLATFORM IMPLICATIONS OF HANDLING A SECOND TYPE OF MEMORY

12.3.1 MULTI-PROCESSOR CACHE COHERENCE

On systems with multiple processors, a directory is used for cache coherence. This directory is implemented as a distributed in-memory directory, with the coherence state of each cache line stored in metadata within the line itself in memory. This implementation improves over the pure snoop-based mechanisms where for each memory access, the processor always checks the other processors' caches to know the coherence state of the line, i.e., if the line is present elsewhere, therefore increasing the latency of each access.

In directory based protocols, the directory tracks if the coherence state is changed. For example, a memory read from another processor is recorded in the metadata in memory. These directory updates result in write to memory (metadata) to record the change in coherence state.

In cases where there are cores in different processors repeatedly reading the same set of lines in the Intel Optane DC Persistent Memory Module, there will be several writes to the Intel Optane DC Persistent Memory Module recording the change in the coherence state each time. These writes are called "directory writes" and tend to be random in nature. As a result, several of these writes can lower the effective Intel Optane DC Persistent Memory Module bandwidth that is available to the application.

From a software standpoint, it is worthwhile to consider how the Intel Optane DC Persistent Memory Module may be accessed by different threads, and if these kind of patterns are observed, one option to consider is to change the coherence protocol for Intel Optane DC Persistent Memory Module regions from directory-based to snoop-based by disabling the directory system-wide.

12.3.2 SHARED QUEUES IN THE MEMORY HIERARCHY

In processors based on the Cascade Lake product, accesses to DRAM and Intel Optane DC Persistent Memory Modules are mostly independent of each other. However, there are instances where both DRAM and memory in Intel Optane DC Persistent Memory Modules reference traverse common paths in the memory subsystem and impact each other.

As an example, there are processor queues that are common between DRAM and Intel Optane DC Persistent Memory Modules. There is a QoS setting in the BIOS that helps arbitrate these queues. Likewise, DRAM and Intel Optane DC Persistent Memory Modules can share the same memory channel (although there are separate queues per channel, the channel itself is shared).

There is a setting to control switching between these separate queues (switching at finer granularity optimizes for latency; switching at coarser granularity allows more bursts and optimizes for bandwidth). These settings are exposed as BIOS knobs; please refer to the BIOS optimization guide for further detail.

12.4 IMPLEMENTING PERSISTENCE FOR MEMORY

In app direct mode, when software is using persistence, software might want to explicitly control that memory stores have been propagated to durability. However, when a processor core issues a write, the data is first combined in a fill buffer and the modified cache line might then be store in the volatile cache of the processors.

For durability, software therefore might need to explicitly evict modified cache lines from the processor caches. This is accomplished by the use of cache line flush instructions (CLFLUSH/CLFLUSHOPT/CLWB). In general, usage of CLFLUSHOPT is recommended over CLFLUSH, as detailed in Section 9.4.6 and Section 9.4.7.

For operations that will reuse the data that is being flushed, usage of CLWB is recommended. CLWB retains a copy of the cache line that is flushed to durability; as a result, for a subsequent access (reuse of data), the line will hit in the caches, reducing the latency of access. If only a small section of large memory ranges are actually modified, it might be worth tracking which sections have been changed and only flush these sections. In particular, if an operation system can track the pages that are written to (“dirty”), and only flushes those pages, it can be more efficient in cases when a small set of the range is written to.

When almost the entire range is written to, it may be more efficient to implement the flushes using optimized code in user-space.

Figure 12-8 shows how msync in Linux* is more efficient when there only a small percentage of files that are dirty, whereas use of CLFLUSHOPT instructions in user space is more efficient when most of the file is dirty.

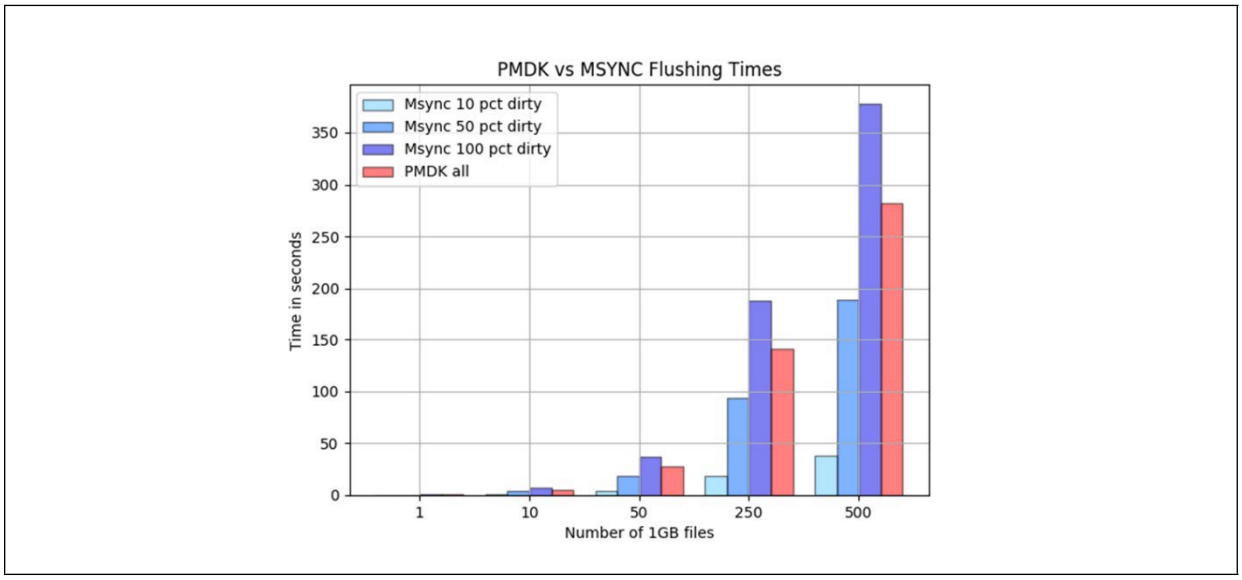


Figure 12-8. PMDK vs. MSYNC Flushing Times[1]

NOTES:

1. When 10% or 50% of the files are dirty, it is more optimal from a software standpoint to use msync instead of flushes in user space. When the file is 100% dirty, the user space implementation (labeled pmdk) is more efficient.

12.5 POWER CONSUMPTION

In general, Intel Optane DC Persistent Memory Module bandwidth is constrained by power consumption, as illustrated with the figure below. If a power limit is imposed by software using techniques like RAPL (Runtime Average Power Limiting), then the overall bandwidth available is appropriately constrained.

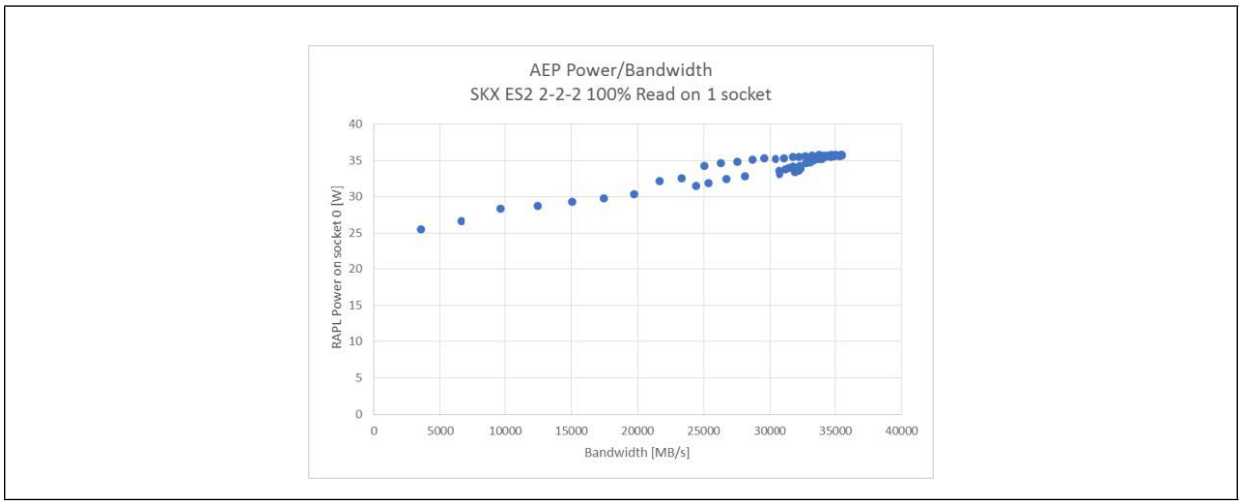


Figure 12-9. Bandwidth vs. Power Consumption

12.5.1 READ-WRITE EQUIVALENCE

On Intel Optane DC Persistent Memory Modules, writes are a lot more expensive in terms of power than reads. As a general guideline, one write consumes as much power as three read operations. This is in stark contrast with DRAM, where the power consumption of reads and writes does not differ as much.

It is therefore clear that writes are to be more sparingly used than reads. This implies that write amplification by software is far more expensive than read amplification when it comes to Intel Optane DC Persistent Memory Module accesses. This is to be considered while designing data structures.

For example, a tree that needs to be rebalanced several times may incur a lot of writes compared to alternate data structures that may have more reads for look up, but fewer writes for insertion.

In the case of a given a power budget, the read-write ratio determines the maximum bandwidth. For example, if several threads are writing to an Intel Optane DC Persistent Memory Module, the power consumed by the threads doing the writes is subtracted from the total power available on the platform.

Figure 12-10 illustrates the read-write equivalence for Intel Optane DC Persistent Memory Module DIMMs within a total power budget.

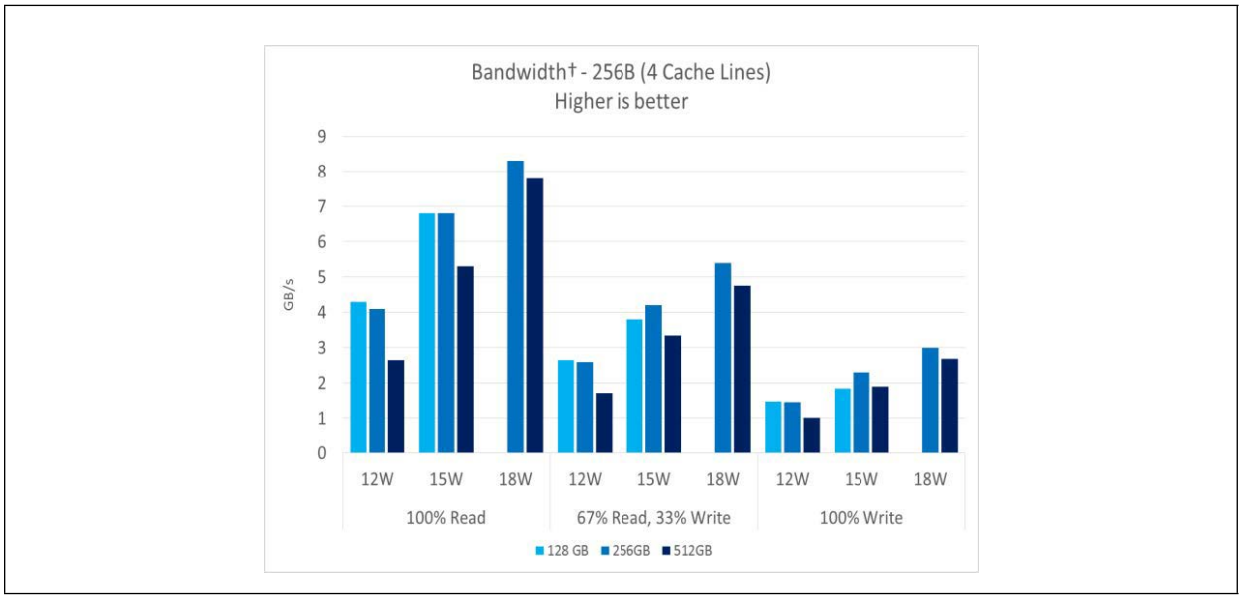


Figure 12-10. Read-Write Equivalence for Intel® Optane™ DC Persistent Memory Module DIMMs within Different Power Budgets[1]

NOTES:

1. The bars on the left show the case for 100% reads. In this scenario, if we consider a power budget of 15W, then ~6.9GB/s of reads are possible. However, if we have the same power budget of 15W, only 2.1GB/s of writes are possible.

12.5.2 SPATIAL AND TEMPORAL LOCALITY

An additional consideration while optimizing for power is to consider the effect of combining data accesses at 256B granularity. The bandwidth available to the application is extremely constrained when there is no locality in accesses, as shown in the figure below.

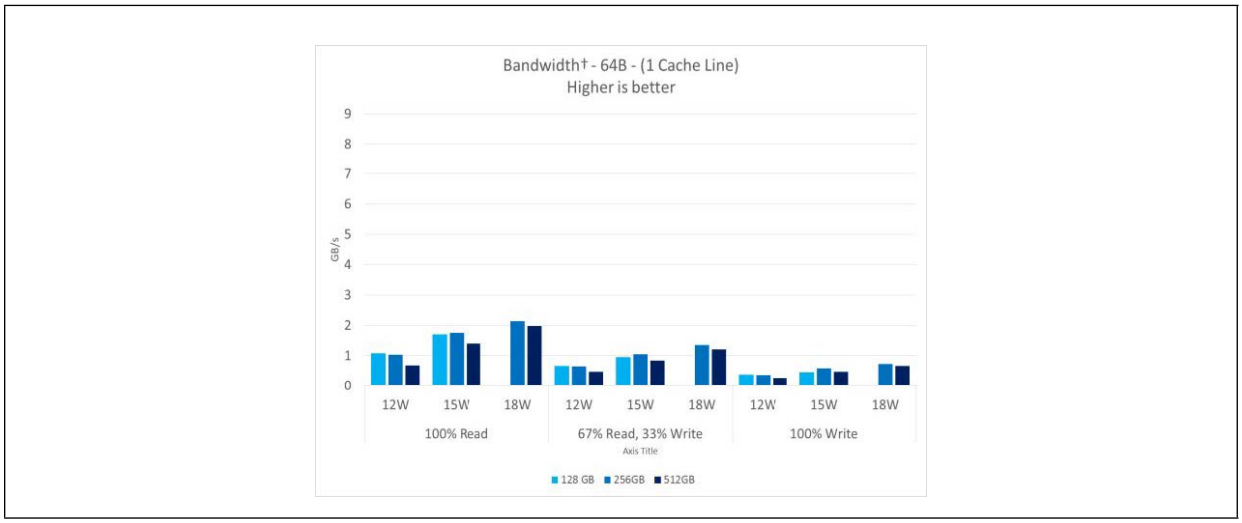


Figure 12-11. Bandwidth Available to Software when There is No Locality at 256B Granularity

From Figure 12-11, we can infer that it is critical to choose data structures that have good access locality at 256B to make good use of a given power budget from a bandwidth standpoint. More specifically, by comparing Figure 12-10 with Figure 12-11, we can observe that using access locality from a 256B window, the bandwidth improves by factors up to 3-4.

CHAPTER 13
64-BIT MODE CODING GUIDELINES

13.1 INTRODUCTION

This chapter describes coding guidelines for application software written to run in 64-bit mode. Some coding recommendations applicable to 64-bit mode are covered in Chapter 3, "General Optimization Guidelines". The guidelines in this chapter should be considered as an addendum to the coding guidelines described in Chapter 3 through Chapter 11, "Multicore and Intel® Hyper-Threading Technology (Intel® HT)".

Software that runs in either compatibility mode or legacy non-64-bit modes should follow the guidelines described in Chapter 3 through Chapter 11.

13.2 CODING RULES AFFECTING 64-BIT MODE

13.2.1 USE LEGACY 32-BIT INSTRUCTIONS WHEN DATA SIZE IS 32 BITS

64-bit mode makes 16 general purpose 64-bit registers available to applications. If application data size is 32 bits, there is no need to use 64-bit registers or 64-bit arithmetic.

The default operand size for most instructions is 32 bits. The behavior of those instructions is to make the upper 32 bits all zeros. For example, when zeroing out a register, the following two instruction streams do the same thing, but the 32-bit version saves one instruction byte:

32-bit version:

xor eax, eax; Performs xor on lower 32bits and zeros the upper 32 bits.

64-bit version:

xor rax, rax; Performs xor on all 64 bits.

This optimization holds true for the lower 8 general purpose registers: EAX, ECX, EBX, EDX, ESP, EBP, ESI, EDI. To access the data in registers R9-R15, the REX prefix is required. Using the 32-bit form there does not reduce code size.

Assembly/Compiler Coding Rule 56. (H impact, M generality) *Use the 32-bit versions of instructions in 64-bit mode to reduce code size unless the 64-bit version is necessary to access 64-bit data or additional registers.*

13.2.2 USE EXTRA REGISTERS TO REDUCE REGISTER PRESSURE

64-bit mode makes eight additional 64-bit general purpose registers and 8 additional XMM registers available to applications. To access the additional registers, a single byte REX prefix is necessary.Eight additional registers can prevent the compiler from needing to spill values onto the stack.

Note that the potential increase in code size, due to the REX prefix, can increase cache misses. This can work against the benefit of using extra registers to access the data. When eight registers are sufficient for an algorithm, don't use the registers that require an REX prefix. This keeps the code size smaller.

Assembly/Compiler Coding Rule 57. (M impact, MH generality) *When they are needed to reduce register pressure, use the 8 extra general purpose registers for integer code and 8 extra XMM registers for floating-point or SIMD code.*

13.2.3 EFFECTIVE USE OF 64-BIT BY 64-BIT MULTIPLICATION

Integer multiplication of 64-bit by 64-bit operands can produce a result that is 128 bits wide. The upper 64 bits of a 128-bit result may take a few cycles longer to be ready than the lower 64 bits. In a dependent chain of addition of integers wider than 128 bits, accessing the high 64-bit result of the multiplication should be delayed relative to the low 64-bit multiplication result for optimal software pipelining.

If the compiler can determine at compile time that the result of a multiplication will not exceed 64 bits, then the compiler should generate the multiplication instruction that produces a 64-bit result. If the compiler or assembly programmer cannot determine that the result will be less than 64 bits, then a multiplication that produces a 128-bit result is necessary.

Assembly/Compiler Coding Rule 58. (ML impact, M generality) *Prefer 64-bit by 64-bit integer multiplication that produces 64-bit results over multiplication that produces 128-bit results.*

Assembly/Compiler Coding Rule 59. (ML impact, M generality) *Stagger accessing the high 64-bit result of a 128-bit multiplication after accessing the low 64-bit results.*

In the Sandy Bridge microarchitecture, the low 64-bit result of a 128-bit multiplication is ready to use in 3 cycles, and the high 64-bit result is ready one cycle after the low 64-bit result. This can speed up the calculation of integer multiplication and division of large integers.

13.2.4 REPLACE 128-BIT INTEGER DIVISION WITH 128-BIT MULTIPLICATION

Modern compilers can transform expressions of integer division in high-level language code with a constant divisor into assembly sequences that use IMUL/MUL to replace IDIV/DIV instructions. Typically, compilers will replace a divisor value that is within the range of 32-bits if the divisor value is known at compile time. If the divisor value is not known at compile time or the divisor is greater than those represented by 32-bits, DIV or IDIV will be generated.

The latency of a DIV instruction with a 128-bit dividend is quite long. For dividend values greater than 64-bits, the latency can range from 70-90 cycles.

The basic technique that a compiler employs to transform integer division into 128-bit integer multiplication is based on the congruence principle of modular arithmetic. It can be easily extended to handle larger divisor values to take advantage of fast 128-bit IMUL/MUL operation.

- The integer equation:

  Dividend = Q * divisor + R, or

  Q = floor(Dividend/Divisor), R = Dividend - Q * Divisor

- Transform to the real number domain:

  floor(Dividend/Divisor) = Dividend/Divisor - R/Divisor, is equivalent to

  Q * C2 = Dividend * (C2 /divisor) + R*C2/Divisor

- One can choose C2 = 2^N to control the rounding of the last term, then

  Q = ((Dividend * (C2 /divisor)) >> N) + ((R*C2/Divisor) >> N).

- If the “divisor” is known at compile time, (C2/Divisor) can be pre-computed into a congruent constant Cx = Ceil(C2/divisor), then the quotient can computed by an integer multiple, followed by a shift:

  Q = (Dividend * Cx) >> N;

  R = Dividend - ((Dividend * Cx) >> N) * divisor;

The 128-bit IDIV/DIV instructions restrict the range of divisor, quotient, and remainder to be within 64- bits to avoid causing numerical exceptions. This presents a challenge for situations with either of the three having a value near the upper bounds of 64-bit and for dividend values nearing the upper bound of 128 bits.

This challenge can be overcome with choosing a larger shift count N, and extending the (Dividend * Cx) operation from the 128-bit range to the next computing-efficient range. For example, if (Dividend * Cx) is greater than 128 bits and N is greater than 63 bits, one can take advantage of computing bits 191:64 of the 192-bit results using 128-bit MUL without implementing a full 192-bit multiplication.

A convenient way to choose the congruent constant Cx is as follows:

- If the range of dividend is within 64 bits: N_{min} ~ BSR(Divisor) + 63.
- In situations of disparate dynamic range of quotient/remainder relative to the range of divisor, raise N accordingly so that quotient/remainder can be computed efficiently.

Consider the computation of quotient/remainder computation for the divisor 10^16 on unsigned dividends near the range of 64-bits. Example 13-1 illustrates using the “MUL r64” instruction to handle a 64-bit dividend with 64-bit divisors.

Example 13-1. Compute 64-bit Quotient and Remainder with 64-bit Divisor

```
_Cx10to16:      ; Congruent constant for 10^16 with shift count ‘N’ = 117
    DD      0c44de15ch      ; floor ( (2^117 / 10^16) + 1)
    DD      0e69594beh      ; Optimize length of Cx to reduce # of 128-bit multiplication
_tento16:                   ; 10^16
    DD      6fc10000h
    DD      002386f2h

    mov     r9, qword ptr [rcx]     ; load 64-bit dividend value
    mov     rax, r9
    mov     rsi, _Cx10to16      ; Congruent Constant for 10^16 with shift count 117
    mul     [rsi]               ; 128-bit multiplication
    mov     r10, qword ptr 8[rsi]   ; load divisor 10^16
    shr     rdx, 53;            ;
    mov     r8, rdx
    mov     rax, r8
    mul     r10         ; 128-bit multiplication
    sub     r9, rax;            ;
    jae     remain
    sub     r8, 1       ; this may be off by one due to round up
    mov     rax, r8

    mul     r10             ; 128-bit multiplication
    sub     r9, rax;            ;
remain:
    mov     rdx, r8         ; quotient
    mov     rax, r9             ; remainder
```

Example 13-2 shows a similar technique to handle a 128-bit dividend with 64-bit divisors.

Example 13-2. Quotient and Remainder of 128-bit Dividend with 64-bit Divisor

```
    mov     rax, qword ptr [rcx]    ; load bits 63:0 of 128-bit dividend from memory
    mov     rsi, _Cx10to16      ; Congruent Constant for 10^16 with shift count 117
    mov     r9, qword ptr [rsi]     ; load Congruent Constant
    mul     r9                  ; 128-bit multiplication
    xor     r11, r11                ; clear accumulator
    mov     rax, qword ptr 8[rcx]   ; load bits 127:64 of 128-bit dividend
    shr     rdx, 53;            ;
    mov     r10, rdx            ; initialize bits 127:64 of 192 b it result
    mul     r9                  ; Accumulate to bits 191:128
    add     rax, r10;           ;
    adc     rdx, r11;           ;
    shr     rax, 53;            ;
    shl     rdx, 11;            ;
    or      rdx, rax;           ;
    mov     r8, qword ptr 8[rsi]    ; load Divisor 10^16
    mov     r9, rdx;            ; approximate quotient, may be off by 1
    mov     rax, r8
    mul     r9                  ; will quotient * divisor > dividend?
    sub     rdx, qword ptr 8[rcx]   ;
    sbb     rax, qword ptr [rcx]    ;

    jb      remain
    sub     r9, 1           ; this may be off by one due to round up
    mov     rax, r8     ; retrieve Divisor 10^16
    mul     r9          ; final quotient * divisor
    sub     rax, qword ptr [rcx]    ;
    sbb     rdx, qword ptr 8[rcx]   ;
remain:
    mov     rdx, r9         ; quotient
    neg     rax             ; remainder
```

The techniques illustrated in Example 13-1 and Example 13-2 can increase the speed of the remainder/quotient calculation of 128-bit dividends to at or below the cost of a 32-bit integer division.

Extending the technique above to deal with a divisor greater than 64-bits is relatively straightforward. One optimization worth considering is to choose a shift count N > 128 bits. This can reduce the number of 128-bit MUL needed to compute the relevant upper bits of (Dividend * Cx).

13.2.5 SIGN EXTENSION TO FULL 64-BITS

When in 64-bit mode, earlier processors can sign-extend to 64 bits in a single micro-op. In 64-bit mode, when the destination is 32 bits, the upper 32 bits must be zeroed.

Zeroing the upper 32 bits requires an extra micro-op and is less optimal than sign extending to 64 bits. While sign extending to 64 bits makes the instruction one byte longer, it reduces the number of micro-ops that the trace cache has to store, improving performance.

For example, to sign-extend a byte into ESI, use:

```
movsx rsi, BYTE PTR[rax]
```

instead of:

```
movsx esi, BYTE PTR[rax]
```

If the next instruction uses the 32-bit form of esi register, the result will be the same. This optimization can also be used to break an unintended dependency. For example, if a program writes a 16-bit value to a register and then writes the register with an 8-bit value, if bits 15:8 of the destination are not needed, use the sign-extended version of writes when available.

- For example:

```
mov    r8w, r9w; Requires a merge to preserve; bits 63:15.
mov    r8b, r10b; Requires a merge to preserve bits 63:8
```

- Can be replaced with:

```
movsx r8, r9w ; If bits 63:8 do not need to be; preserved.
movsx r8, r10b  ; If bits 63:8 do not need to; be preserved.
```

In the above example, the moves to R8W and R8B both require a merge to preserve the rest of the bits in the register. There is an implicit real dependency on R8 between the 'MOV R8W, R9W' and 'MOV R8B, R10B'. Using MOVSX breaks the real dependency and leaves only the output dependency, which the processor can eliminate through renaming.

For processors based on Intel Core microarchitecture, zeroing the upper 32 bits is faster than sign-extend to 64 bits. For processors based on Nehalem microarchitecture, zeroing or sign-extend the upper bits is single micro-op.

13.3 ALTERNATE CODING RULES FOR 64-BIT MODE

13.3.1 USE 64-BIT REGISTERS INSTEAD OF TWO 32-BIT REGISTERS FOR 64-BIT ARITHMETIC RESULT

Legacy 32-bit mode offers the ability to support extended precision integer arithmetic (such as 64-bit arithmetic). However, 64-bit mode offers native support for 64-bit arithmetic. When 64-bit integers are desired, use the 64-bit forms of arithmetic instructions.

In 32-bit legacy mode, getting a 64-bit result from a 32-bit by 32-bit integer multiply requires three registers; the result is stobbred in 32-bit chunks in the EDX:EAX pair. When the instruction is available in 64-bit mode, using the 32-bit version of the instruction is not the optimal implementation if a 64-bit result is desired. Use the extended registers.

For example, the following code sequence loads the 32-bit values sign-extended into the 64-bit registers and performs a multiply:

```
movsx rax, DWORD PTR[x]
movsx rcx, DWORD PTR[y]
imul  rax, rcx
```

The 64-bit version above is more efficient than using the following 32-bit version:

```
mov eax, DWORD PTR[x]
mov ecx, DWORD PTR[y]
imul ecx
```

In the 32-bit case above, EAX is required to be a source. The result ends up in the EDX:EAX pair instead of in a single 64-bit register.

Assembly/Compiler Coding Rule 60. (ML impact, M generality) *Use the 64-bit versions of multiply for 32-bit integer multiplies that require a 64 bit result.*

To add two 64-bit numbers in 32-bit legacy mode, the add instruction followed by the addc instruction is used. For example, to add two 64-bit variables (X and Y), the following four instructions could be used:

```
mov eax, DWORD PTR[X]
mov edx, DWORD PTR[X+4]
add eax, DWORD PTR[Y]
adc edx, DWORD PTR[Y+4]
```

The result will end up in the two-register EDX:EAX.

In 64-bit mode, the above sequence can be reduced to the following:

```
mov rax, QWORD PTR[X]
add rax, QWORD PTR[Y]
```

The result is stored in rax. One register is required instead of two.

Assembly/Compiler Coding Rule 61. (ML impact, M generality) *Use the 64-bit versions of add for 64-bit adds.*

13.3.2 USING SOFTWARE PREFETCH

Intel recommends that software developers follow the recommendations in Chapter 3 and Chapter 9 when considering the choice of organizing data access patterns to take advantage of the hardware prefetcher (versus using software prefetch).

Assembly/Compiler Coding Rule 62. (L impact, L generality) *If software prefetch instructions are necessary, use the prefetch instructions provided by SSE.*

CHAPTER 14
INTEL® SSE4.2 AND SIMD PROGRAMMING FOR TEXT-PROCESSING/LEXING/PARSING

String/text processing spans a discipline that often employs techniques different from traditional SIMD integer vector processing. Much of the traditional string/text algorithms are character based, where characters may be represented by encodings (or code points) of fixed or variable byte sizes. Textual data represents a vast amount of raw data and often carrying contextual information.

The contextual information embedded in raw textual data often requires algorithmic processing dealing with a wide range of attributes, such as:

- Character values.
- Character positions.
- Character encoding formats.
- Subsetting of character sets.
- Strings of explicit or implicit lengths.
- Tokens.
- Delimiters.

Contextual objects may be represented by sequential characters within a predefined character subset (e.g. decimal-valued strings). Textual streams may contain embedded state transitions separating objects of different contexts (e.g., tag-delimited fields).

Traditional Integer SIMD vector instructions may, in some simpler situations, be successful to speed up simple string processing functions. Intel SSE4.2 includes four new instructions that offer advances to computational algorithms targeting string/text processing, lexing and parsing of either unstructured or structured textual data.

14.1 INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS

Intel SSE4.2 provides four instructions that can accelerate string and text processing by combining the efficiency of SIMD programming techniques and the embedded lexical primitives:

- PCMPESTRI
- PCMPESTRM
- PCMPISTRI
- PCMPISTRM

Simple examples of these instructions include:

- String length determination.
- Direct string comparison.
- String case handling.
- Delimiter/token processing.
- Locating word boundaries.
- Locating sub-string matches in large text blocks.

Sophisticated application of Intel SSE4.2 can accelerate XML parsing and Schema validation.

The processor's support for Intel SSE4.2 is indicated by the feature flag value returned in ECX [bit 20] after executing CPUID instruction with EAX input value of 1 (i.e. SSE4.2 is supported if CPUID.01H:ECX.SSE4_2 [bit 20] = 1). Therefore, software must verify CPUID.01H:ECX.SSE4_2 [bit 20] is set before using these 4 instructions. (Verifying CPUID.01H:ECX.SSE4_2 = 1 is also required before using PCMPGTQ or CRC32. Verifying CPUID.01H:ECX.POPCNT[Bit 23] = 1 is required before using the POPCNT instruction.)

These string/text processing instructions work by performing up to 256 comparison operations on text fragments. Each text fragment can be sixteen bytes. They can handle fragments of different formats: either byte or word elements. Each of these four instructions can be configured to perform four types of parallel comparison operation on two text fragments.

The aggregated intermediate result of a parallel comparison of two text fragments become a bit patterns:sixteen bits for processing byte elements or eight bits for word elements. These instruction provide additional flexibility, using bit fields in the immediate operand of the instruction syntax, to configure an unary transformation (polarity) on the first intermediate result.

Lastly, the instruction's immediate operand offers a output selection control to further configure the flexibility of the final result produced by the instruction. The rich configurability of these instruction is summarized in Figure 14-1.

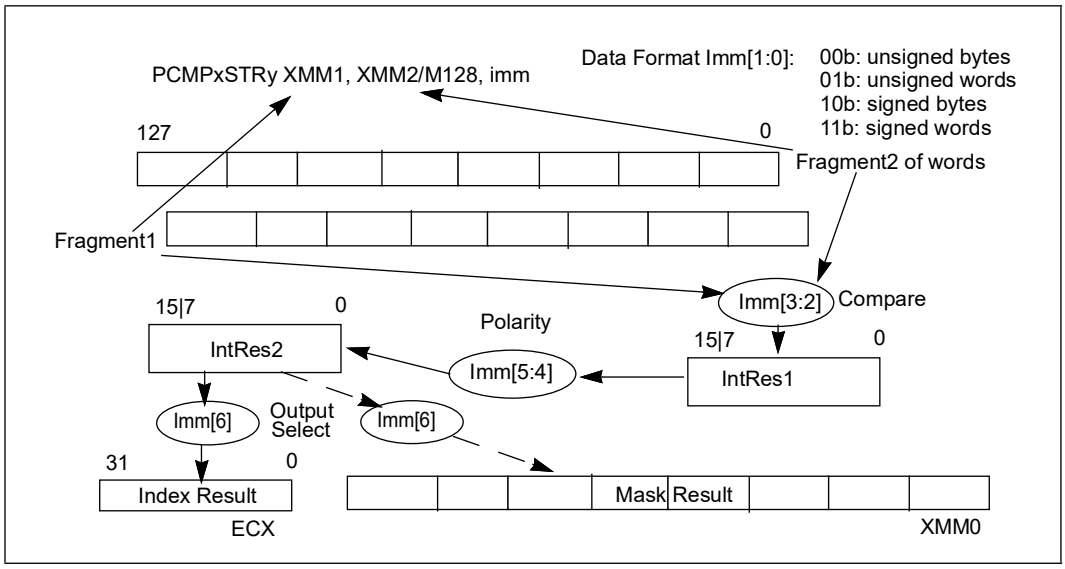


Figure 14-1. Intel® SSE4.2 String/Text Instruction Immediate Operand Control

The **PCMPxSTRI** instructions produce final result as an integer index in ECX, the PCMPxSTRM instructions produce final result as a bit mask in the XMM0 register. The PCMPISTRy instructions support processing string/text fragments using implicit length control via null termination for handling string/text of unknown size. the PCMPESTRy instructions support explicit length control via EDX:EAX register pair to specify the length text fragments in the source operands.

The first intermediate result, IntRes1, is an aggregated result of bit patterns from parallel comparison operations done on pairs of data elements from each text fragment, according to the imm[3:2] bit field encoding, see Table 14-1.

Table 14-1. Intel® SSE4.2 String/Text Instructions Compare Operation on N-Elements

Imm[3:2]	Name	IntRes1[i] is TRUE if	Potential Usage
00B	Equal Any	Element i in fragment2 matches any element j in fragment1	Tokenization, XML parser
01B	Ranges	Element i in fragment2 is within any range pairs specified in fragment1	Subsetting, Case handling, XML parser, Schema validation
10B	Equal Each	Element i in fragment2 matches element i in fragment1	Strcmp()
11B	Equal Ordered	Element i and subsequent, consecutive valid elements in fragment2 match fully or partially with fragment1 starting from element 0	Substring Searches, KMP, Strstr()

Input data element format selection using imm[1:0] can support signed or unsigned byte/word elements.

The bit field imm[5:4] allows applying a unary transformation on IntRes1, see Table 14-2.

Table 14-2. Intel® SSE4.2 String/Text Instructions Unary Transformation on IntRes1

Imm[5:4]	Name	IntRes2[i] =	Potential Usage
00B	No Change	IntRes1[i]	-
01B	Invert	-IntRes1[i]	-
10B	No Change	IntRes1[i]	-
11B	Mask Negative	IntRes1[i] if element i of fragment2 is invalid, otherwise - IntRes1[i]	-

The output selection field, imm[6] is described in Table 14-3.

Table 14-3. Intel® SSE4.2 String/Text Instructions Output Selection Imm[6]

Imm[6]	Instruction	Final Result	Potential Usage
0B	PCMPxSTRI	ECX = offset of least significant bit set in IntRes2 if IntRes2 != 0, otherwise ECX = number of data element per 16 bytes	-
0B	PCMPxSTRM	XMM0 = ZeroExtend(IntRes2);	-
1B	PCMPxSTRI	ECX = offset of most significant bit set in IntRes2 if IntRes2 != 0, otherwise ECX = number of data element per 16 bytes	-
1B	PCMPxSTRM	Data element i of XMM0 = SignExtend(IntRes2[i]);	-

The comparison operation on each data element pair is defined in Table 14-4. This table defines the type of comparison operation between valid data elements in the last row and boundary conditions when the fragment in a source operand may contain invalid data elements (rows one through three). Arithmetic comparison are performed only if both data elements are valid element in fragment1 and fragment2, as shown in row four.

Table 14-4. SSE4.2 String/Text Instructions Element-Pair Comparison Definition

Fragment1 Element	Fragment2 Element	Imm[3:2]= 00B, Equal Any	Imm[3:2]= 01B, Ranges	Imm[3:2]= 10B, Equal Each	Imm[3:2]= 11B, Equal Ordered
Invalid	Invalid	Force False	Force False	Force True	Force True
Invalid	Valid	Force False	Force False	Force False	Force True
Valid	Invalid	Force False	Force False	Force False	Force False
Valid	Valid	Compare	Compare	Compare	Compare

The string and text processing instruction provides several aid to handle end-of-string situations, see Table 14-5. Additionally, the PCMPxSTRy instructions are designed to not require 16-byte alignment to simplify text processing requirements.

Table 14-5. SSE4.2 String/Text Instructions Eflags Behavior

EFLAGs	Description	Potential Usage
CF	Reset if IntRes2 = 0; Otherwise set	When CF=0, ECX= #of data element to scan next
ZF	Reset if entire 16-byte fragment2 is valid	likely end-of-string
SF	Reset if entire 16-byte fragment1 is valid	
OF	IntRes2[0];	

14.1.1 CRC32

CRC32 instruction computes the 32-bit cyclic redundancy checksum signature for byte/word/dword or qword stream of data. It can also be used as a hash function. For example, a dictionary uses hash indices to de-reference strings. CRC32 instruction can be easily adapted for use in this situation.

Example 14-1 shows a straight forward hash function that can be used to evaluate the hash index of a string to populate a hash table. Typically, the hash index is derived from the hash value by taking the remainder of the hash value modulo the size of a hash table.

Example 14-1. A Hash Function Examples

```
unsigned int hash_str(unsigned char* pStr)
{  unsigned int hVal = (unsigned int)(*pStr++);
   while (*pStr)
  {    hVal = (hashVal * CONST_A) + (hVal >> 24) + (unsigned int)(*pStr++);
  }
  return hVal;
}
```

CRC32 instruction can be use to derive an alternate hash function. Example 14-2 takes advantage the 32-bit granular CRC32 instruction to update signature value of the input data stream. For string of small to moderate sizes, using the hardware accelerated CRC32 can be twice as fast as Example 14-1.

Example 14-2. Hash Function Using CRC32

```
static unsigned cn_7e = 0x7efefeff, Cn_81 = 0x81010100;

unsigned int hash_str_32_crc32x(unsigned char* pStr)
{  unsigned  *pDW = (unsigned  *) &pStr[1];
   unsigned  short *pWd = (unsigned short *) &pStr[1];
   unsigned int tmp, hVal = (unsigned int)(*pStr);
   if( !pStr[1]) ;
   else {
      tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
      while ( !tmp ) // loop until there is byte in *pDW had 0x00
      {
         hVal = _mm_crc32_u32 (hVal, *pDW ++);
         tmp = ((pDW[0] +cn_7e ) ^(pDW[0]^ -1)) & Cn_81;
      };
      if(!pDW[0]);
      else  if(pDW[0] < 0x100) { // finish last byte that's non-zero
         hVal = _mm_crc32_u8 (hVal, pDW[0]);
      }

      else  if(pDW[0] < 0x10000) { // finish last two byte that's non-zero
         hVal = _mm_crc32_u16 (hVal, pDW[0]);
      }
      else  { // finish last three byte that's non-zero
         hVal = _mm_crc32_u32 (hVal, pDW[0]);
      }
   }
   return hVal;
}
```

14.2 USING INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS

String libraries provided by high-level languages or as part of system library are used in a wide range of situations across applications and privileged system software. These situations can be accelerated using a replacement string library that implements PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM.

Although system-provided string library provides standardized string handling functionality and interfaces, most situations dealing with structured document processing requires considerable more sophistication, optimization, and services unavailable from system-provided string libraries. For example, structured document processing software often architect different class objects to provide building block functionality to service specific needs of the application. Often application may choose to disperse equivalent string library services into separate classes (string, lexer, parser) or integrate memory management capability into string handling/lexing/parsing objects.

PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions are general-purpose primitives that software can use to build replacement string libraries or build class hierarchy to provide lexing/parsing services for structured document processing. XML parsing and schema validation are examples of the latter situations.

Unstructured, raw text/string data consist of characters, and have no natural alignment preferences. Therefore, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions are architected to not require the 16-Byte alignment restrictions of other 128-bit SIMD integer vector processing instructions.

With respect to memory alignment, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM support unaligned memory loads like other unaligned 128-bit memory access instructions, e.g. MOVDQU.

Unaligned memory accesses may encounter special situations that require additional coding techniques, depending on the code running in ring 3 application space or in privileged space. Specifically, an unaligned 16-byte load may cross page boundary. Section 14.2.1 discusses a technique that application code can use. Section 14.2.2 discusses the situation string library functions needs to deal with. Section 14.3 gives detailed examples of using PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM instructions to implement equivalent functionality of several string library functions in situations that application code has control over memory buffer allocation.

14.2.1 UNALIGNED MEMORY ACCESS AND BUFFER SIZE MANAGEMENT

In application code, the size requirements for memory buffer allocation should consider unaligned SIMD memory semantics and application usage.

For certain types of application usage, it may be desirable to make distinctions between valid buffer range limit versus valid application data size (e.g. a video frame). The former must be greater or equal to the latter.

To support algorithms requiring unaligned 128-bit SIMD memory accesses, memory buffer allocation by a caller function should consider adding some pad space so that a callee function can safely use the address pointer safely with unaligned 128-bit SIMD memory operations.

The minimal padding size should be the width of the SIMD register that might be used in conjunction with unaligned SIMD memory access.

14.2.2 UNALIGNED MEMORY ACCESS AND STRING LIBRARY

String library functions may be used by application code or privileged code. String library functions must be careful not to violate memory access rights. Therefore, a replacement string library that employ SIMD unaligned access must employ special techniques to ensure no memory access violation occur.

Section 14.3.6 provides an example of a replacement string library function implemented with Intel SSE4.2 and demonstrates a technique to use 128-bit unaligned memory access without unintentionally crossing page boundary.

14.3 INTEL® SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES

Software implementing Intel SSE4.2 instruction must use CPUID feature flag mechanism to verify processor's support for SSE4.2. Details can be found in:

- Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1
- CPUID of Chapter 3, "Instruction Set Reference, A-L" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

In the following sections, we use several examples in string/text processing of progressive complexity to illustrates the basic techniques of adapting the SIMD approach to implement string/text processing using PCMPxSTRy instructions in Intel SSE4.2.

For simplicity, we will consider string/text in byte data format in situations that caller functions have allocated sufficient buffer size to support unaligned 128-bit SIMD loads from memory without encountering side-effects of cross page boundaries.

14.3.1 NULL CHARACTER IDENTIFICATION (STRLEN EQUIVALENT)

The most widely used string function is probably strlen(). One can view the lexing requirement of strlen() is to identify the null character in a text block of unknown size (end of string condition). Brute-force, byte-granular implementation fetches data inefficiently by loading one byte at a time.

Optimized implementation using general-purpose instructions can take advantage of dword operations in 32-bit environment (and qword operations in 64-bit environment) to reduce the number of iterations.

A 32-bit assembly implementation of strlen() is shown Example 14-3. The peak execution throughput of handling EOS condition is determined by eight ALU instructions in the main loop.

Example 14-3. Strlen() Using General-Purpose Instructions

```
int strlen_asm(const  char* s1)
{int len = 0;
  _asm{
  mov  ecx, s1
  test   ecx, 3 ; test addr aligned to dword
  je     short _main_loop1  ; dword aligned loads would be faster
_malign_str1:
  mov  al, byte ptr [ecx] ; read one byte at a time
  add   ecx, 1
  test   al, al ; if we find a null, go calculate the length
  je     short _byte3a
test ecx, 3; test if addr is now aligned to dword
  jne    short _malign_str1; if not, repeat
  align16
_main_loop1:; read each 4-byte block and check for a NULL char in the dword
  mov  eax, [ecx]; read 4 byte to reduce loop count
  mov  edx, 7efefeffh
  add   edx, eax
  xor    eax, -1
  xor    eax, edx
  add   ecx, 4; increment address pointer by 4
  test   eax, 81010100h ; if no null code in 4-byte stream, do the next 4 bytes
  je     short _main_loop1
  ;      there is a null char in the dword we just read,
  ;      since we already advanced pointer ecx by 4, and the dword is lost
  mov  eax, [ecx -4]; re-read the dword that contain at least a null char
  test   al, al     ; if byte0 is null
  je     short _byte0a; the least significant byte is null
  test   ah, ah    ; if byte1 is null
  je     short _byte1a
  test   eax, 00ff0000h; if byte2 is null
  je     short _byte2a
```

Example 14-3. Strlen() Using General-Purpose Instructions (Contd.)

```
test eax, 00ff000000h; if byte3 is null
  je     short _byte3a
  jmp  short _main_loop1
_byte3a:
; we already found the null, but pointer already advanced by 1
  lea    eax, [ecx-1]; load effective address corresponding to null code
  mov  ecx, s1
  sub   eax, ecx; difference between null code and start address
  jmp  short _resulta
_byte2a:
lea  eax, [ecx-2]
  mov  ecx, s1
  sub   eax, ecx
  jmp  short _resulta
_byte1a:
  lea    eax, [ecx-3]
  mov  ecx, s1
  sub   eax, ecx
  jmp  short _resulta
_byte0a:
  lea    eax, [ecx-4]
  mov  ecx, s1
  sub   eax, ecx
_resulta:
  mov    len, eax; store result
  }
  return len;
}
```

The equivalent functionality of EOS identification can be implemented using PCMPISTRI. Example 14-4 shows a simplistic Intel SSE4.2 implementation to scan a text block by loading 16-byte text fragments and locate the null

termination character. Example 14-5 shows the optimized Intel SSE4.2 implementation that demonstrates the effectiveness of memory disambiguation to improve instruction-level parallelism.

Example 14-4. Sub-optimal PCMPISTRI Implementation of EOS handling

```
static char ssch2[16]= {0x1, 0xff, 0x00, }; // range values for non-null characters
int strlen_un_optimized(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor    ecx, ecx ; initial offset to 0
_loopc:
  add   eax, ecx   ; update addr pointer to start of text fragment
  pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
      ; if there is a null char in the 16Byte fragment at [eax], zf will be set.
      ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
  jnz    short _loopc; xmm1 has no null code, ecx has 16, continue search
  ;       we have a null code in xmm1, ecx has the offset of the null code i
  add   eax, ecx       ; add ecx to the address of the last fragment2/xmm1
  mov  edx, s1; retrieve effective address of the input string
sub  eax, edx;the string length
  mov  len, eax; store result
  }
return len;
}
```

The code sequence shown in Example 14-4 has a loop consisting of three instructions. From a performance tuning perspective, the loop iteration has loop-carry dependency because address update is done using the result (ECX value) of a previous loop iteration. This loop-carry dependency deprives the out-of-order engine's capability to have multiple iterations of the instruction sequence making forward progress. The latency of memory loads, the latency of these instructions, any bypass delay could not be amortized by OOO execution in the presence of loop-carry dependency.

A simple optimization technique to eliminate loop-carry dependency is shown in Example 14-5.

Using memory disambiguation technique to eliminate loop-carry dependency, the cumulative latency exposure of the three-instruction sequence of Example 14-5 is amortized over multiple iterations, the net cost of executing each iteration (handling sixteen bytes) is less than three cycles. In contrast, handling 4=four bytes of string data using eight ALU instructions in Example 14-3 will also take a little less than three cycles per iteration. Whereas each iteration of the code sequence in Example 14-4 will take more than ten cycles because of loop-carry dependency.

Example 14-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency

```
int strlen_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor    ecx, ecx ; initial offset to 0
  sub   eax, 16 ; address arithmetic to eliminate extra instruction and a branch
```

Example 14-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency (Contd.)

```
_loopc:
  add   eax, 16 ; adjust address pointer and disambiguate load address for each iteration
  pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
      ; if there is a null char in [eax] fragment, zf will be set.
      ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
  jnz short _loopc ; ECX will be 16 if there is no null byte in [eax], so we disambiguate
_endofstring:
  add   eax, ecx       ; add ecx to the address of the last fragment
  mov  edx, s1; retrieve effective address of the input string
  sub   eax, edx;the string length
  mov  len, eax; store result
  }
  return len;
}
```

Tuning Suggestion 4. (H impact, H generality) *Loop-carry dependency that depends on the ECX result of PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM for address adjustment must be minimized. Isolate code paths that expect ECX result will be 16 (bytes) or 8 (words), replace these values of ECX with constants in address adjustment expressions to take advantage of memory disambiguation hardware.*

14.3.2 WHITE-SPACE-LIKE CHARACTER IDENTIFICATION

Character-granular-based text processing algorithms have developed techniques to handle specific tasks to remedy the efficiency issue of character-granular approaches. One such technique is using look-up tables for character subset classification. For example, some application may need to separate alpha-numeric characters from white-space-like characters. More than one character may be treated as white-space characters.

Example 14-6 illustrates a simple situation of identifying white-space-like characters for the purpose of marking the beginning and end of consecutive non-white-space characters.

Example 14-6. WordCnt() Using C and Byte-Scanning Technique

```
// Counting words involves locating the boundary of contiguous non-whitespace characters.
// Different software may choose its own mapping of white space character set.
// This example employs a simple definition for tutorial purpose:
// Non-whitespace character set will consider: A-Z, a-z, 0-9, and the apostrophe mark '
// The example uses a simple technique to map characters into bit patterns of square waves
// we can simply count the number of falling edges

static char alphnrange[16]= {0x27, 0x27, 0x30, 0x39, 0x41, 0x5a, 0x61, 0x7a, 0x0};
static char alp_map8[32] = {0x0, 0x0, 0x0, 0x0, 0x80,0x0,0xff, 0x3,0xfe, 0xff, 0xff, 0x7, 0xfe, 0xff, 0xff, 0x7}; // 32
byte lookup table, 1s map to bit patterns of alpha numerics in alphnrange
int wordcnt_c(const  char* s1)
{int  i, j, cnt = 0;
char cc, cc2;
char  flg[3]; // capture the a wavelet to locate a falling edge
  cc2 = cc = s1[0];
  // use the compacted bit pattern to consolidate multiple comparisons into one look up
  if( alp_map8[cc>>3] & ( 1<< ( cc & 7) ) )
  {   flg[1] = 1; } // non-white-space char that is part of a word,
  // we're including apostrophe in this example since counting the
  // following 's' as a separate word would be kind of silly
  else
  {   flg[1] = 0; }  // 0: whitespace, punctuations not be considered as part of a word

  i = 1; // now we're ready to scan through the rest of the block
  // we'll try to pick out each falling edge of the bit pattern to increment word count.
  // this works with consecutive white spaces, dealing with punctuation marks, and
  // treating hyphens as connecting two separate words.
  while (cc2 )
{  cc2 = s1[i];
      if( alp_map8[cc2>>3] & ( 1<< ( cc2 & 7) ) )
      {   flg[2] = 1;} // non-white-space
      else
      {   flg[2] = 0;} // white-space-like

      if( !flg[2] && flg[1] )
      {   cnt ++; }// found the falling edge
      flg[1] = flg[2];
      i++
  }
  return cnt;
}
```

In Example 14-6, a 32-byte look-up table is constructed to represent the ascii code values 0x0-0xff, and partitioned with each bit of 1 corresponding to the specified subset of characters. While this bit-lookup technique simplifies the comparison operations, data fetching remains byte-granular.

Example 14-7 shows an equivalent implementation of counting words using PCMPISTRM. The loop iteration is performed at 16-byte granularity instead of byte granularity. Additionally, character set subsetting is easily expressed using range value pairs and parallel comparisons between the range values and each byte in the text fragment are performed by executing PCMPISTRI once.

Example 14-7. WordCnt() Using PCMPISTRM

```
// an SSE4.2 example of counting words using the definition of non-whitespace character
// set of {A-Z, a-z, 0-9, '}. Each text fragment (up to 16 bytes) are mapped to a
// 16-bit pattern, which may contain one or more falling edges. Scanning bit-by-bit
// would be inefficient and goes counter to leveraging SIMD programming techniques.
// Since each falling edge must have a preceding rising edge, we take a finite
// difference approach to derive a pattern where each rising/falling edge maps to 2-bit pulse,
// count the number of bits in the 2-bit pulses using popcnt and divide by two.
int wdcnt_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm3, alphnrange ; load range value pairs to detect non-white-space codes
  xor    ecx, ecx
  xor    esi, esi
  xor    edx, edx

  movdquxmm1, [eax]
  pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
  movdqa xmm4, xmm0
  movdqa     xmm1, xmm0
  psrld xmm4, 15    ; save MSB to use in next iteration
  movdqa xmm5, xmm1
  psllw xmm5, 1; lsb is effectively mapped to a white space
  pxor  xmm5, xmm0; the first edge is due to the artifact above
  pextrd edi, xmm5, 0
  jz      _lastfragment; if xmm1 had a null, zf would be set
  popcnt edi, edi; the first fragment will include a rising edge
  add     esi, edi
  mov  ecx, 16
_loopc:
  add   eax, ecx ; advance address pointer
  movdquxmm1, [eax]
  pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
  movdqa xmm5, xmm4  ; retrieve the MSB of the mask from last iteration
  movdqa xmm4, xmm0
  psrld xmm4, 15    ; save mSB of this iteration for use in next iteration
  movdqa xmm1, xmm0
```

Example 14-7. WordCnt() Using PCMPISTRM (Contd.)

```
   psllw xmm1, 1
   por   xmm5, xmm1  ; combine MSB of last iter and the rest from current iter
   pxor   xmm5, xmm0; differentiate binary wave form into pattern of edges
   pextrdedi, xmm5, 0 ; the edge patterns has (1 bit from last, 15 bits from this round)
   jz      _lastfragment; if xmm1 had a null, zf would be set
   mov  ecx, 16; xmm1, had no null char, advance 16 bytes
   popcntedi, edi; count both rising and trailing edges
 add esi, edi; keep a running count of both edges
   jmp   short _loopc
_lastfragment:
   popcntedi, edi; count both rising and trailing edges
   add   esi, edi; keep a running count of both edges
   shr    esi, 1; word count corresponds to the trailing edges
   mov  len, esi
   }
   return len;
}
```

14.3.3 SUBSTRING SEARCHES

Strstr() is a common function in the standard string library. Typically, A library may implement strstr(sTarg, sRef) with a brute-force, byte-granular technique of iterative comparisons between the reference string with a round of string comparison with a subset of the target string. Brute-force, byte-granular techniques provide reasonable efficiency when the first character of the target substring and the reference string are different, allowing subsequent string comparisons of target substrings to proceed forward to the next byte in the target string.

When a string comparison encounters partial matches of several characters (i.e. the sub-string search found a partial match starting from the beginning of the reference string) and determined the partial match led to a false-match. The brute-force search process need to go backward and restart string comparisons from a location that had participated in previous string comparison operations. This is referred to as re-trace inefficiency of the brute-force substring search algorithm. See Figure 14-2.

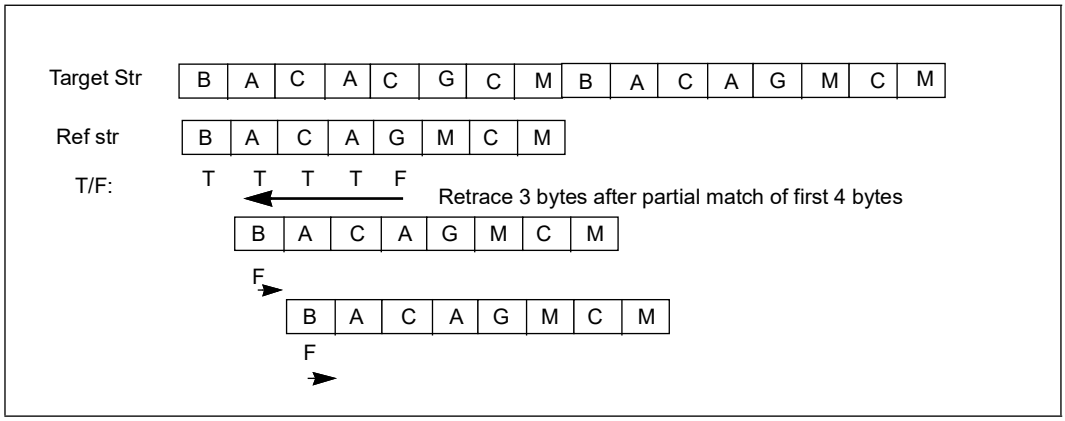


Figure 14-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search

The Knuth, Morris, Pratt algorithm[1] (KMP) provides an elegant enhancement to overcome the re-trace inefficiency of brute-force substring searches. By deriving an overlap table that is used to manage retrace distance when a partial

match leads to a false match, KMP algorithm is very useful for applications that search relevant articles containing keywords from a large corpus of documents.

Example 14-8 illustrates a C-code example of using KMP substring searches.

Example 14-8 also includes the calculation of the KMP overlap table. Typical usage of KMP algorithm involves multiple

Example 14-8. KMP Substring Search in C

```
// s1 is the target string of length cnt1
// s2 is the reference string of length cnt2
// j is the offset in target string s1 to start each round of string comparison
// i is the offset in reference string s2 to perform byte granular comparison

int str_kmp_c(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int i, j;
    i = 0; j = 0;
    while ( i+j < cnt1) {
        if( s2[i] == s1[i+j]) {
            i++;
            if( i == cnt2) break; // found full match
        }

        else {
            j = j+i - ovrlap_tbl[i]; // update the offset in s1 to start next round of string compare
            if( i > 0) {
                i = ovrlap_tbl[i]; // update the offset of s2 for next string compare should start at
            }
        }
void kmp_precalc(const char * s2, int cnt2)
{int i = 2;
char nch = 0;
  ovrlap_tbl[0] = -1; ovrlap_tbl[1] = 0;
     // pre-calculate KMP table
  while( i < cnt2) {
     if( s2[i-1] == s2[nch]) {
        ovrlap_tbl[i] =  nch +1;
        i++; nch++;
     }
     else if ( nch > 0) nch = ovrlap_tbl[nch];
     else {
        ovrlap_tbl[i] = 0;
        i++;
     }
  };
  ovrlap_tbl[cnt2] = 0;
}
```

invocation of the same reference string, so the overhead of precalculating the overlap table is easily amortized. When

1. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt; SIAM J. Comput. Volume 6, Issue 2, pp. 323-350 (1977)

a false match is determined at offset i of the reference string, the overlap table will predict where the next round of string comparison should start (updating the offset j), and the offset in the reference string that byte-granular character comparison should resume/restart.

While KMP algorithm provides efficiency improvement over brute-force byte-granular substring search, its best performance is still limited by the number of byte-granular operations. To demonstrate the versatility and built-in lexical capability of PCMPISTRI, we show an Intel SSE4.2 implementation of substring search using brute-force 16-byte granular approach in Example 14-9, and combining KMP overlap table with substring search using PCMPISTRI in Example 14-10.

Example 14-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic

```
int strsubs_sse4_2i(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
  if( cnt2 > cnt1 || !cnt1) return -1;
  frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
  frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment


  while(rcnt1 > 0)
  {   cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
      cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
      if( !cmp) { // we have a partial match that needs further analysis
          if( cmp_s) { // if we're done with s2
              if( pt)
              {   idx = (int) ((char *) pt - (char *) s1) ; }
              else
              {   idx = (int) ((char *) p1 - (char *) s1) ;  }
              return idx;
          }
          // we do a round of string compare to verify full match till end of s2
          if( pt == NULL) pt = p1;
          cmp2 = 16;
          rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
          while(  cmp2 == 16  && rcnt2) {  // each 16B frag matches,
              rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
              rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
              if( rcnt1 <=0 || rcnt2 <= 0 ) break;
              p1 = (__m128i *)(((char *)p1) + 16);
              p2 = (__m128i *)(((char *)p2) + 16);
              frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
              frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
              cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
```

Example 14-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic (Contd.)

```
};
        if( !rcnt2 || rcnt2 == cmp2) {
            idx = (int) ((char *) pt - (char *) s1) ;
            return idx;
        }
        else if ( rcnt1 <= 0) { // also cmp2 < 16, non match
        if( cmp2 == 16 && ((rcnt1 + 16) >= (rcnt2+16) ) )
        {idx = (int) ((char *) pt - (char *) s1) ;
        return idx;

        }
            else return -1;
        }

        else { // in brute force, we advance fragment offset in target string s1 by 1
            p1 = (__m128i *)(((char *)pt) + 1);  // we're  not taking advantage of kmp
            rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
            pt = NULL;
            p2 = (__m128i *)((char *)s2) ;
            rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
            frag1 = _mm_loadu_si128(p1);// load next fragment from s1
            frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
        }
      }
      else{
        if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
        else p1 = (__m128i *)(((char *)p1) + cmp);
        rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
        if( pt && cmp ) pt = NULL;
        frag1 = _mm_loadu_si128(p1);// load next fragment from s1
      }
    }
    return idx;
}
```

In Example 14-9, address adjustment using a constant to minimize loop-carry dependency is practised in two places:

- In the inner while loop of string comparison to determine full match or false match (the result cmp2 is not used for address adjustment to avoid dependency).
- In the last code block when the outer loop executed PCMPISTRI to compare sixteen sets of ordered compare between a target fragment with the first 16-byte fragment of the reference string, and all sixteen ordered compare operations produced false result (producing cmp with a value of 16).

Example 14-10 shows an equivalent intrinsic implementation of substring search using Intel SSE4.2 and KMP overlap table. When the inner loop of string comparison determines a false match, the KMP overlap table is consulted to determine the address offset for the target string fragment and the reference string fragment to minimize retrace.

Notably, a significant portions of retrace with a retrace distance less than 15 bytes are avoided even in the brute-force Intel SSE4.2 implementation of Example 14-9. This is due to the order-compare primitive of PCMPISTRI. “Ordered compare” performs sixteen sets of string fragment compare, and many false match with less than fifteen bytes of partial matches can be filtered out in the same iteration that executed PCMPISTRI.

Retrace distance of greater than fifteen bytes does not get filtered out by the Example 14-9. By consulting with the KMP overlap table, Example 14-10 can eliminate retraces of greater than fifteen bytes.

Example 14-10. Substring Search Using PCMPISTRI and KMP Overlap Table

```
int strkmp_sse4_2(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;

int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
  if( cnt2 > cnt1 || !cnt1) return -1;
  frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
  frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment

  while(rcnt1 > 0)
  {   cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
      cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
      if( !cmp) { // we have a partial match that needs further analysis
          if( cmp_s) { // if we've reached the end with s2
              if( pt)
              {   idx = (int) ((char *) pt - (char *) s1) ; }
              else
              {   idx = (int) ((char *) p1 - (char *) s1) ;  }
              return idx;
          }
      // we do a round of string compare to verify full match till end of s2
          if( pt == NULL) pt = p1;
          cmp2 = 16;
          rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);

          while(  cmp2 == 16 && rcnt2) {  // each 16B frag matches
                    rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
                    rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
                    if( rcnt1 <=0 || rcnt2 <= 0 ) break;
                    p1 = (__m128i *)(((char *)p1) + 16);
                    p2 = (__m128i *)(((char *)p2) + 16);
                    frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
                    frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                    cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb,
eq each
          };
          if( !rcnt2 || rcnt2 == cmp2) {
                idx = (int) ((char *) pt - (char *) s1) ;
                        return idx;
          }
          else if ( rcnt1 <= 0 ) { // also cmp2 < 16, non match
              return -1;
          }
          else { // a partial match led to false match, consult KMP overlap table for addr adjustment
                kpm_i = (int) ((char *)p1 - (char *)pt)+ cmp2 ;
                p1 = (__m128i *)(((char *)pt) + (kpm_i - ovrlap_tbl[kpm_i]));  // use kmp to skip retrace
```

Example 14-10. Substring Search Using PCMPISTRI and KMP Overlap Table (Contd.)

```
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                pt = NULL;
                p2 = (__m128i *)(((char *)s2) + (ovrlap_tbl[kpm_i]));
                rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
            }
        }

        else{
            if( kpm_i && ovrlap_tbl[kpm_i]) {
                p2 = (__m128i *)(((char *)s2) );
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                //p1 = (__m128i *)(((char *)p1) );

                //rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                if( pt && cmp ) pt = NULL;
                rcnt2 = cnt2  ;
                //frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                kpm_i = 0;
            }
            else { // equ order comp resulted in sub-frag match or non-match
            if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
            else p1 = (__m128i *)(((char *)p1) + cmp);
            rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
            if( pt && cmp ) pt = NULL;
            frag1 = _mm_loadu_si128(p1);// load next fragment from s1
            }
        }
    }
    return idx;
}
```

The relative speed up of byte-granular KMP, brute-force Intel SSE4.2, and Intel SSE4.2 with KMP overlap table over byte-granular brute-force substring search is illustrated in the graph that plots relative speedup over percentage of retrace for a reference string of 55 bytes long. A retrace of 40% in the graph meant, after a partial match of the first 22 characters, a false match is determined.

So when brute-force, byte-granular code has to retrace, the other three implementation may be able to avoid the need to retrace because:

- Example 14-8 can use KMP overlap table to predict the start offset of next round of string compare operation after a partial-match/false-match, but forward movement after a first-character-false-match is still byte-granular.
- Example 14-9 can avoid retrace of shorter than 15 bytes but will be subject to retrace of 21 bytes after a partial-match/false-match at byte 22 of the reference string. Forward movement after each order-compare-false-match is 16 byte granular.
- Example 14-10 avoids retrace of 21 bytes after a partial-match/false-match, but KMP overlap table lookup incurs some overhead. Forward movement after each order-compare-false-match is 16 byte granular.

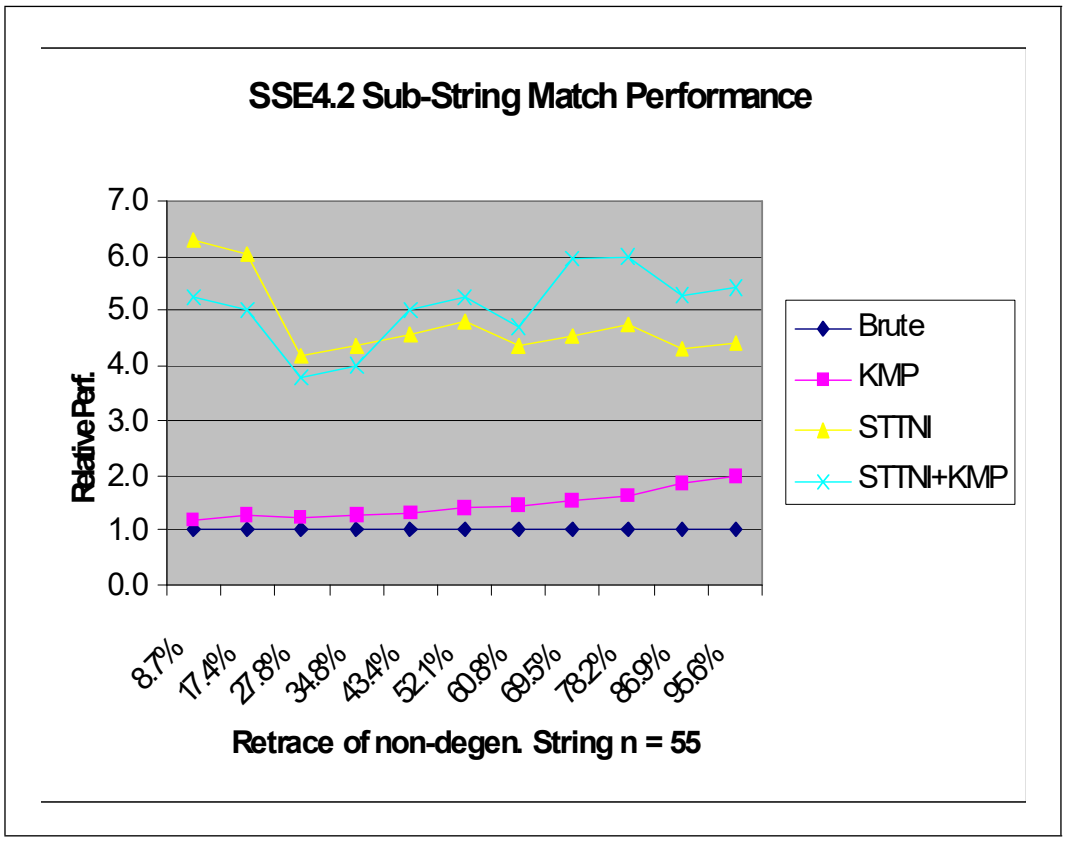


Figure 14-3. Intel® SSE4.2 Speedup of SubString Searches

14.3.4 STRING TOKEN EXTRACTION AND CASE HANDLING

Token extraction is a common task in text/string handling. It is one of the foundation of implementing lexer/parser objects of higher sophistication. Indexing services also build on tokenization primitives to sort text data from streams.

Tokenization requires the flexibility to use an array of delimiter characters.

A library implementation of Strtok_s() may employ a table-lookup technique to consolidate sequential comparisons of the delimiter characters into one comparison (similar to Example 14-6). An SSE4.2 implementation of the equivalent functionality of strtok_s() using intrinsic is shown in Example 14-11.

Example 14-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic

```
char ws_map8[32]; // packed bit lookup table for delimiter characters
char * strtok_sse4_2i(char* s1, char *sdlm, char ** pCtxt)
{
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, stmpz, stmp1;
int cmp_z, jj =0;
int start, endtok, s_idx, ldx;
  if (sdlm == NULL || pCtxt == NULL) return NULL;
  if( p1 == NULL && *pCtxt == NULL) return NULL;
     if( s1 == NULL)  {
     if( *pCtxt[0] == 0 ) { return NULL; }
```

Example 14-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic (Contd.)

```
      p1 = (__m128i *) *pCtxt;
      s1 = *pCtxt;
   }
   else p1 = (__m128i *) s1;
   memset(&ws_map8[0], 0, 32);
   while (sdlm[jj] ) {
      ws_map8[ (sdlm[jj] >> 3) ] |= (1 << (sdlm[jj] & 7) ); jj ++
   }
frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
   stmpz  = _mm_loadu_si128((__m128i *)sdelimiter);
   // if the first char is not a delimiter , proceed to check non-delimiter,
   // otherwise need to skip leading delimiter chars
   if( ws_map8[s1[0]>>3] & (1 <<  (s1[0]&7)) ) {
    start = s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// unsigned bytes/equal any, invert, lsb
   }
   else start = s_idx = 0;

// check if we're dealing with short input string less than 16 bytes
   cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
   if( cmp_z) {  // last fragment
      if( !start) {
         endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);
         if( endtok == 16) { // didn't find delimiter at the end, since it's null-terminated
            // find where is the null byte
            *pCtxt = s1+ 1+ _mm_cmpistri(frag1, frag1, 0x40);
            return s1;
         }
         else { // found a delimiter that ends this word
         s1[start+endtok] = 0;
         *pCtxt = s1+start+endtok+1;
         }
      }

      else {
         if(!s1[start] ) {
            *pCtxt = s1 + start +1;
            return NULL;
         }
         p1 = (__m128i *)(((char *)p1) + start);
         frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
         endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
         if( endtok == 16) { // looking for delimiter, found none
            *pCtxt = (char *)p1 + 1+ _mm_cmpistri(frag1, frag1, 0x40);
            return s1+start;
         }
         else { // found delimiter before null byte
            s1[start+endtok] = 0;
            *pCtxt = s1+start+endtok+1;
         }
      }
   }
```

Example 14-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic (Contd.)

```
  else
  {   while ( !cmp_z && s_idx == 16) {
          p1 = (__m128i *)(((char *)p1) + 16);
          frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
          s_idx = _mm_cmpistri(stmpz, frag1, 0x10);// unsigned bytes/equal any, invert, lsb
          cmp_z = _mm_cmpistrz(stmpz, frag1, 0x10);
      }
      if(s_idx != 16) start = ((char *) p1  -s1) + s_idx;
      else { // corner case if we ran to the end looking for delimiter and never found a non-dilimiter
          *pCtxt = (char *)p1  +1+ _mm_cmpistri(frag1, frag1, 0x40);
          return NULL;
      }
      if( !s1[start] ) { // in case a null byte follows delimiter chars
          *pCtxt = s1 + start+1;
          return NULL;
      }
      // now proceed to find how many non-delimiters are there
      p1 = (__m128i *)(((char *)p1) + s_idx);
      frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
      endtok = ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
      cmp_z = 0;
      while ( !cmp_z && ldx == 16) {
          p1 = (__m128i *)(((char *)p1) + 16);
          frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
          ldx = _mm_cmpistri(stmpz, frag1, 0x00);// unsigned bytes/equal any, lsb
          cmp_z = _mm_cmpistrz(stmpz, frag1, 0x00);
          if(cmp_z) { endtok += ldx; }
      }

      if( cmp_z ) { // reached the end of s1
          if(  ldx < 16) // end of word found by finding a delimiter
              endtok += ldx;
          else {  // end of word found by finding the null
              if( s1[start+endtok]) // ensure this frag don't start with null byte
              endtok += 1+ _mm_cmpistri(frag1, frag1, 0x40);
          }
      }
      *pCtxt = s1+start+endtok+1;
      s1[start+endtok] = 0;
  }
  return (char *) (s1+ start);
}
```

An Intel SSE4.2 implementation of the equivalent functionality of strupr() using intrinsic is shown in Example 14-12.

Example 14-12. I Equivalent Strupr() Using PCMPISTRM Intrinsic

```
static char uldelta[16]= {0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20};
static char ranglc[6]= {0x61, 0x7a, 0x00, 0x00, 0x00, 0x00};
char * strup_sse4_2i(  char* s1)
{int len = 0, res = 0;
__m128i *p1 = (__m128i *) s1;
__m128ifrag1, ranglo, rmsk, stmpz, stmp1;
int cmp_c, cmp_z, cmp_s;
  if( !s1[0]) return (char *) s1;
  frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
  ranglo = _mm_loadu_si128((__m128i *)ranglc);// load up to 16 bytes of fragment
  stmpz  = _mm_loadu_si128((__m128i *)uldelta);

  cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);// range compare, produce byte masks
  while (!cmp_z)
  {
     rmsk = _mm_cmpistrm(ranglo, frag1, 0x44); // producing byte mask
     stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk);  // bytes of lc preserved, other bytes replaced by const
     stmp1 =_mm_sub_epi8(stmp1, stmpz);  // bytes of lc becomes uc, other bytes are now zero
     stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); //bytes of lc replaced by uc, other bytes unchanged
     _mm_storeu_si128(p1, stmp1);//
     p1 = (__m128i *)(((char *)p1) + 16);
     frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
     cmp_z = _mm_cmpistrz(ranglo, frag1, 0x44);
  }

  if( *(char *)p1 == 0) return (char *) s1;
  rmsk = _mm_cmpistrm(ranglo, frag1, 0x44);// byte mask, valid lc bytes are 1, all other 0
  stmp1 = _mm_blendv_epi8(stmpz, frag1, rmsk);  // bytes of lc continue, other bytes replaced by const
  stmp1 =_mm_sub_epi8(stmp1, stmpz);  // bytes of lc becomes uc, other bytes are now zero
  stmp1 = _mm_blendv_epi8(frag1, stmp1, rmsk); //bytes of lc replaced by uc, other bytes unchanged
  rmsk = _mm_cmpistrm(frag1, frag1, 0x44);// byte mask, valid  bytes are 1, invalid bytes are zero
  _mm_maskmoveu_si128(stmp1, rmsk, (char *) p1);//
  return (char *) s1;
}
```

14.3.5 UNICODE PROCESSING AND PCMPXSTRY

Unicode representation of string/text data is required for software localization. UTF-16 is a common encoding scheme for localized content. In UTF-16 representation, each character is represented by a code point. There are two classes of code points: 16-bit code points and 32-bit code points which consists of a pair of 16-bit code points in specified value range, the latter is also referred to as a surrogate pair.

A common technique in unicode processing uses a table-loop up method, which has the benefit of reduced branching. As a tutorial example we compare the analogous problem of determining properly-encoded UTF-16 string length using general purpose code with table-lookup vs. Intel SSE4.2.

Example 14-13 lists the C code sequence to determine the number of properly-encoded UTF-16 code points (either 16-bit or 32-bit code points) in a unicode text block. The code also verifies if there are any improperly-encoded surrogate pairs in the text block.

Example 14-13. UTF16 VerStrlen() Using C and Table Lookup Technique

```
// This example demonstrates validation of surrogate pairs (32-bit code point) and
// tally the number of16-bit and 32-bit code points in the text block
// Parameters: s1 is pointer to input utf-16 text block.
// pLen: store count of utf-16 code points
// return the number of 16-bit code point encoded in the surrogate range but do not form
// a properly encoded surrogate pair. if 0: s1 is a properly encoded utf-16 block,
// If return value >0 then s1 contains invalid encoding of surrogates

int u16vstrlen_c(const  short* s1, unsigned * pLen)
{int  i, j, cnt = 0, cnt_invl = 0, spcnt= 0;
unsigned short cc, cc2;
char  flg[3];

 cc2 = cc = s1[0];
 // map each word in s1into bit patterns of 0, 1or 2 using a table lookup
// the first half of a surrogate pair must be encoded between D800-DBFF and mapped as 2
// the 2nd half of a surrogate pair must be encoded between DC00-DFFF and mapped as 1
// regular 16-bit encodings are mapped to 0, except null code mapped to 3
flg[1] = utf16map[cc];
 flg[0] = flg[1];
 if(!flg[1]) cnt ++;
 i = 1;

 while (cc2 ) // examine each non-null word encoding
 { cc2 = s1[i];
   flg[2] = utf16map[cc2];
   if( (flg[1] && flg[2] && (flg[1]-flg[2] == 1) )  )
   {   spcnt ++; }// found a surrogate pair
   else if(flg[1] == 2 && flg[2] != 1)
   {  cnt_invl += 1; } // orphaned 1st half
   else if( !flg[1]  && flg[2] == 1)
   {  cnt_invl += 1; } // orphaned 2nd half
   else
   {   if(!flg[2]) cnt ++;// regular non-null code16-bit code point
       else ;
   }
   flg[0] = flg[1];// save the pair sequence for next iteration
   flg[1] = flg[2];
   i++;
 }
 *pLen = cnt + spcnt;
 return cnt_invl;
}
```

The VerStrlen() function for UTF-16 encoded text block can be implemented using SSE4.2.

Example 14-14 shows the listing of Intel SSE4.2 assembly implementation and Example 14-15 shows the listing of Intel SSE4.2 intrinsic listings of VerStrlen().

Example 14-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI

```
// complementary range values for detecting either halves of 32-bit UTF-16 code point
static short ssch0[16]= {0x1, 0xd7ff, 0xe000, 0xffff, 0, 0};
// complementary range values for detecting the 1st half of 32-bit UTF-16 code point
static short ssch1[16]= {0x1, 0xd7ff, 0xdc00, 0xffff, 0, 0};
// complementary range values for detecting the 2nd half of 32-bit UTF-16 code point
static short ssch2[16]= {0x1, 0xdbff, 0xe000, 0xffff, 0, 0};


int utf16slen_sse4_2a(const  short* s1, unsigned * pLen)
{int len = 0, res = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch0 ; load range value to identify either halves
  movdquxmm3, ssch1 ; load range value to identify 1st half (0xd800 to 0xdbff)
  movdquxmm4, ssch2 ; load range value to identify 2nd half (0xdc00 to 0xdfff)
  xor   ecx, ecx
  xor   edx, edx; store # of 32-bit code points (surrogate pairs)
  xor   ebx, ebx; store # of non-null 16-bit code points
  xor   edi, edi ; store # of invalid word encodings


_loopc:
  shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
  add   eax, ecx
  movdquxmm1, [eax] ; load a string fragment of up to 8 words
  pcmpistri  xmm2, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
      ; if there is a utf-16 null wchar in xmm1, zf will be set.
      ; if all 8 words in the comparison matched range,
      ;  none of bits in the intermediate result will be set after polarity inversions,
      ;  and ECX will return with a value of 8
  jz      short _lstfrag; if null code, handle last fragment
  ; if ecx < 8, ecx point to a word of either 1st or 2nd half of a 32-bit code point
  cmp  ecx, 8
  jne   _chksp
  add   ebx, ecx   ; accumulate # of 16-bit non-null code points
  mov  ecx, 8 ; ecx must be 8 at this point, we want to avoid loop carry dependency
  jmp   _loopc
_chksp:; this fragment has word encodings in the surrogate value range
  add   ebx, ecx ; account for the 16-bit code points
  shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
  add   eax, ecx
  movdquxmm1, [eax] ; ensure the fragment start with word encoding in either half
  pcmpistri  xmm3, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
  jz      short _lstfrag2; if null code, handle the last fragment
cmpecx, 0 ; properly encoded 32-bit code point must start with 1st half
  jg      _invalidsp; some invalid s-p code point exists in the fragment
  pcmpistri  xmm4, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
  cmp  ecx, 1 ; the 2nd half must follow the first half
  jne   _invalidsp
```

Example 14-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI (Contd.)

```
   add   edx, 1; accumulate # of valid surrogate pairs
   add   ecx, 1 ; we want to advance two words
   jmp   _loopc
_invalidsp:; the first word of this fragment is either the 2nd half or an un-paired 1st half
   add   edi, 1  ; we have an invalid code point (not a surrogate pair)
   mov  ecx, 1 ; advance one word and continue scan for 32-bit code points
   jmp   _loopc
_lstfrag:
   add   ebx, ecx ; account for the non-null 16-bit code points
_morept:
   shl    ecx, 1; pcmpistri with word processing return ecx in word granularity, multiply by 2 to get byte offset
   add   eax, ecx
   mov  si, [eax] ; need to check for null code
   cmp  si, 0
   je     _final
   movdquxmm1, [eax] ; load remaining word elements which start with either 1st/2nd half
   pcmpistri  xmm3, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
_lstfrag2:
   cmp  ecx, 0 ; a valid 32-bit code point must start from 1st half
   jne    _invalidsp2
   pcmpistri  xmm4, xmm1, 15h; unsigned words, ranges, invert, lsb index returned to ecx
   cmp  ecx, 1
   jne    _invalidsp2

   add   edx, 1
   mov  ecx, 2
   jmp   _morept
_invalidsp2:
   add   edi, 1
   mov  ecx, 1
   jmp   _morept
_final:
   add   edx, ebx; add # of 16-bit and 32-bit code points
   mov  ecx, pLen; retrieve address of pointer provided by caller
   mov  [ecx], edx; store result of string length to memory
   mov  res, edi
   }
   return res;
}
```

Example 14-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI

```
int utf16slen_i(const  short* s1, unsigned * pLen)
{int len = 0, res = 0;
__m128i *pF = (__m128i *) s1;
__m128iu32 =_mm_loadu_si128((__m128i *)ssch0);
__m128i  u32a =_mm_loadu_si128((__m128i *)ssch1);
__m128i  u32b =_mm_loadu_si128((__m128i *)ssch2);
__m128ifrag1;

int offset1 = 0, cmp, cmp_1, cmp_2;
intcnt_16 = 0, cnt_sp=0, cnt_invl= 0;
short *ps;
while (1) {
  pF = (__m128i *)(((short *)pF) + offset1);
     frag1 = _mm_loadu_si128(pF);// load up to 8 words
     // does frag1 contain either halves of a 32-bit UTF-16 code point?
     cmp = _mm_cmpistri(u32, frag1, 0x15);// unsigned bytes, equal order, lsb index returned to ecx
  if (_mm_cmpistrz(u32, frag1, 0x15))// there is a null code in frag1
     {   cnt_16 += cmp;
         ps = (((short *)pF) + cmp);
         while (ps[0])
         {   frag1 = _mm_loadu_si128( (__m128i *)ps);
             cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
             if(!cmp_1)
             {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
                 if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
                 else {cnt_invl++;  offset1= 1;}
             }
else
             {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
                 if(!cmp_2) {cnt_invl ++; offset1 = 1;}
                 else {cnt_16 ++; offset1 = 1; }
             }
             ps = (((short *)ps) + offset1);
         }
         break;
     }
     if(cmp != 8) // we have at least some half of 32-bit utf-16 code points
     {   cnt_16 += cmp; // regular 16-bit UTF16 code points
         pF = (__m128i *)(((short *)pF) + cmp);
         frag1 = _mm_loadu_si128(pF);
         cmp_1 = _mm_cmpistri(u32a, frag1, 0x15);
         if(!cmp_1)
         {   cmp_2 = _mm_cmpistri(u32b, frag1, 0x15);
             if( cmp_2 ==1) { cnt_sp++; offset1 = 2;}
     else {cnt_invl++;  offset1= 1;}
         }
     else {cnt_invl ++; offset1 = 1;
         }
     }
```

Example 14-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI (Contd.)

```
  else {
        offset1 = 8; // increment address by 16 bytes to handle next fragment
        cnt_16+= 8;
    }
};
    *pLen = cnt_16 + cnt_sp;
    return cnt_invl;
}
```

14.3.6 REPLACEMENT STRING LIBRARY FUNCTION USING INTEL® SSE4.2

Unaligned 128-bit SIMD memory access can fetch data cross page boundary, since system software manages memory access rights with page granularity.

Implementing a replacement string library function using SIMD instructions must not cause memory access violation. This requirement can be met by adding a small amounts of code to check the memory address of each string fragment. If a memory address is found to be within 16 bytes of crossing over to the next page boundary, string processing algorithm can fall back to byte-granular technique.

Example 14-16 shows an Intel SSE4.2 implementation of strcmp() that can replace byte-granular implementation supplied by standard tools.

Example 14-16. Replacement String Library Strcmp Using Intel® SSE4.2

```
// return 0 if strings are equal, 1 if greater, -1 if less
int strcmp_sse4_2(const char *src1, const char *src2)
{
  int val;
  __asm{
    mov     esi, src1 ;
    mov     edi, src2
    mov     edx, -16 ; common index relative to base of either string pointer
    xor     eax, eax
topofloop:
    add     edx, 16 ; prevent loop carry dependency
next:
    lea     ecx, [edi+edx] ; address of fragment that we want to load
    and     ecx, 0x0fff ; check least significant12 bits of addr for page boundary
    cmp     ecx, 0x0ff0
    jg      too_close_pgb ; branch to byte-granular if within 16 bytes of boundary
    lea     ecx, [edi+edx] ; do the same check for each fragment of 2nd string
    and     ecx, 0x0fff
    cmp     ecx, 0x0ff0
    jg      too_close_pgb
    movdquxmm2, BYTE PTR[edi+edx]
    movdquxmm1, BYTE PTR[edi+edx]
    pcmpistrixmm2, xmm1, 0x18 ; equal each
ja      topofloop
    jnc     ret_tag
    add     edx, ecx ; ecx points to the byte offset that differ
```

Example 14-16. Replacement String Library Strcmp Using Intel® SSE4.2 (Contd.)

```
  not_equal:
      movzx   eax, BYTE PTR[esi+edx]
      movzx   edx, BYTE PTR[edi+edx]
      cmp     eax, edx
      cmova   eax, ONE
      cmovb   eax, NEG_ONE
      jmp     ret_tag
too_close_pgb:
      add     edx, 1 ; do byte granular compare
      movzx   ecx, BYTE PTR[esi+edx-1]
      movzx   ebx, BYTE PTR[edi+edx-1]
      cmp     ecx, ebx
      jne     inequality
      add     ebx, ecx
      jnz     next
      jmp     ret_tag
inequality:
      cmovb   eax, NEG_ONE
      cmova   eax, ONE
ret_tag:
      mov     [val], eax
  }
  return(val);
}
```

In Example 14-16, eight instructions were added following the label “next” to perform 4KByte boundary checking of address that will be used to load two string fragments into registers. If either address is found to be within sixteen bytes of crossing over to the next page, the code branches to byte-granular comparison path following the label “too_close_pgb“.

The return values of Example 14-16 uses the convention of returning 0, +1, -1 using CMOV. It is straight forward to modify a few instructions to implement the convention of returning 0, positive integer, negative integer.

14.4 INTEL® SSE4.2-ENABLED NUMERICAL AND LEXICAL COMPUTATION

Intel SSE4.2 can enable SIMD programming techniques to explore byte-granular computational problems that were considered unlikely candidates for using SIMD instructions. We consider a common library function atol() in its full 64-bit flavor of converting a sequence of alpha numerical characters within the range representable by the data type __int64.

There are several attributes of this string-to-integer problem that poses as difficult challenges for using prior SIMD instruction sets (before the introduction of Intel SSE4.2) to accelerate the numerical computation aspect of string-to-integer conversions:

- Character subset validation: Each character in the input stream must be validated with respect to the character subset definitions and conform to data representation rules of white space, signs, numerical digits. Intel SSE4.2 provides the perfect tools for character subset validation.
- State-dependent nature of character validation: While SIMD computation instructions can expedite the arithmetic operations of “multiply by 10 and add“, the arithmetic computation requires the input byte stream to consist of numerical digits only. For example, the validation of numerical digits, white-space, and the

presence/absence of sign, must be validated in mid-stream. The flexibility of the SSE4.2 primitive can handle these state-dependent validation well.

- Additionally, exit condition to wrap up arithmetic computation can happen in mid-stream due to invalid characters, or due to finite representable range of the data type (~10^19 for int64, no more than 10 non-zero-leading digits for int32) may lead one to believe this type data stream consisting of short bursts are not suited for exploring SIMD ISA and be content with byte-granular solutions.

Because of the character subset validation and state-dependent nature, byte-granular solutions of the standard library function tends to have a high start-up cost (for example, converting a single numerical digit to integer may take 50 or 60 cycles), and low throughput (each additional numeric digit in the input character stream may take 6-8 cycles per byte).

A high level pseudo-operation flow of implementing a library replacement of atol() is described in Example 14-17.

Example 14-17. High-level flow of Character Subset Validation for String Conversion

1. Check Early_Out Exit Conditions (e.g. first byte is not valid).
2. Check if 1st byte is white space and skip any additional leading white space.
3. Check for the presence of a sign byte.
4. Check the validity of the remaining byte stream if they are numeric digits.
5. If the byte stream starts with '0', skip all leading digits that are '0'.
6. Determine how many valid non-zero-leading numeric digits.
7. Convert up to 16 non-zero-leading digits to int64 value.
8. load up to the next 16 bytes safely and check for consecutive numeric digits
9. Normalize int64 value converted from the first 16 digits, according to # of remaining digits,
10. Check for out-of-bound results of normalized intermediate int64 value,
11. Convert remaining digits to int64 value and add to normalized intermediate result,
12. Check for out-of-bound final results.

Example 14-18 shows the code listing of an equivalent functionality of atol() capable of producing int64 output range. Auxiliary function and data constants are listed in Example 14-19.

Example 14-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI

```
__int64 sse4i_atol(const  char* s1)
{char  *p = ( char *) s1;
 int NegSgn = 0;
 __m128i mask0;
 __m128i  value0, value1;
 __m128i  w1, w1_l8, w1_u8, w2, w3 = _mm_setzero_si128();
 __int64 xxi;
 int index, cflag, sflag, zflag, oob=0;
// check the first character is valid via lookup
if ( (BtMLValDecInt[ *p >> 3] & (1 << ((*p) & 7)) ) == 0) return 0;
// if the first character is white space, skip remaining white spaces
if (BtMLws[*p >>3] & (1 <<((*p) & 7)) )
{  p ++;
  value0 = _mm_loadu_si128 ((__m128i *) listws);
skip_more_ws:
mask0 = __m128i_strloadu_page_boundary (p);
```

Example 14-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI (Contd.)

```
  /* look for the 1st non-white space character */
    index = _mm_cmpistri (value0, mask0,  0x10);
    cflag = _mm_cmpistrc (value0, mask0,   0x10);
    sflag = _mm_cmpistrs (value0, mask0,   0x10);
    if( !sflag && !cflag)
    {  p = ( char *) ((size_t) p + 16);
      goto skip_more_ws;
    }
    else       p = (  char *) ((size_t) p + index);
}
if( *p == '-')
{  p++;
  NegSgn = 1;
}
else if( *p == '+') p++;

/* load up to 16 byte safely and check how many valid numeric digits we can do SIMD */
value0 = _mm_loadu_si128 ((__m128i *) rangenumint);
mask0 = __m128i_strloadu_page_boundary (p);
index = _mm_cmpistri (value0, mask0,  0x14);
zflag = _mm_cmpistrz (value0, mask0,  0x14);
/* index points to the first digit that is not a valid numeric digit */
if( !index) return 0;
else if (index == 16)
{  if( *p == '0') /* if all 16 bytes are numeric digits */
  {  /* skip leading zero */
    value1 = _mm_loadu_si128 ((__m128i *) rangenumintzr);
    index = _mm_cmpistri (value1, mask0,  0x14);
    zflag = _mm_cmpistrz (value1, mask0,  0x14);
    while(index == 16 && !zflag )
    {  p = ( char *) ((size_t) p + 16);
      mask0 = __m128i_strloadu_page_boundary (p);
      index = _mm_cmpistri (value1, mask0,  0x14);
      zflag = _mm_cmpistrz (value1, mask0,  0x14);
    }
/* now the 1st digit is non-zero, load up to 16 bytes and update index  */
    if( index < 16)
      p = ( char *) ((size_t) p + index);
    /* load up to 16 bytes of non-zero leading numeric digits */
    mask0 = __m128i_strloadu_page_boundary (p);
    /* update index to point to non-numeric character or indicate we may have more than 16 bytes */
    index = _mm_cmpistri (value0, mask0,  0x14);
  }
}
if( index == 0)  return 0;
else if( index == 1)   return (NegSgn? (long long) -(p[0]-48): (long long) (p[0]-48));
// Input digits in xmm are ordered in reverse order. the LS digit of output is next to eos
// least sig numeric digit aligned to byte 15 , and subtract 0x30 from each ascii code
mask0  = ShfLAlnLSByte( mask0, 16 -index);
w1_u8 = _mm_slli_si128 ( mask0, 1);
```

Example 14-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI (Contd.)

```
w1 = _mm_add_epi8( mask0, _mm_slli_epi16 (w1_u8, 3)); /* mul by 8  and add */
w1 = _mm_add_epi8( w1, _mm_slli_epi16 (w1_u8, 1)); /* 7 LS bits per byte, in bytes 0, 2, 4, 6, 8, 10, 12, 14*/
w1 = _mm_srli_epi16( w1, 8);  /* clear out upper bits of each wd*/
w2 = _mm_madd_epi16(w1, _mm_loadu_si128(  (__m128i *) &MulplyPairBaseP2[0]) ); /* multiply base^2, add
adjacent word,*/
w1_u8 = _mm_packus_epi32 ( w2, w2);  /* pack 4 low word of each dword into 63:0 */
w1 = _mm_madd_epi16(w1_u8, _mm_loadu_si128(  (__m128i *) &MulplyPairBaseP4[0]) ); /* multiply base^4,
add adjacent word,*/
w1 = _mm_cvtepu32_epi64( w1);  /*   converted dw was in 63:0, expand to qw */
w1_l8 = _mm_mul_epu32(w1, _mm_setr_epi32(  100000000, 0, 0, 0) );
w2 = _mm_add_epi64(w1_l8, _mm_srli_si128 (w1, 8) );
if( index < 16)
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
/* 64-bit integer allow up to 20 non-zero-leading digits. */
/* accumulate each 16-digit fragment*/
w3 = _mm_add_epi64(w3, w2);
/* handle next batch of up to 16 digits, 64-bit integer only allow 4 more digits */
p = ( char *) ((size_t) p + 16);
if( *p == 0)
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
mask0 = __m128i_strloadu_page_boundary (p);
/* index points to first non-numeric digit */
index = _mm_cmpistri (value0, mask0,  0x14);
zflag = _mm_cmpistrz (value0, mask0,  0x14);
if( index  == 0) /* the first char is not valid numeric digit */
{  xxi = _mm_extract_epi64(w2, 0);
   return (NegSgn? (long long) -xxi: (long long) xxi);
}
if ( index > 3)  return (NegSgn? (long long) RINT64VALNEG: (long long) RINT64VALPOS);
/* multiply low qword by base^index */
w1 =  _mm_mul_epu32( _mm_shuffle_epi32( w2,  0x50), _mm_setr_epi32( MulplyByBaseExpN
    [index - 1] , 0, MulplyByBaseExpN[index-1], 0));
w3 = _mm_add_epi64(w1, _mm_slli_epi64 (_mm_srli_si128(w1, 8), 32 ) );
mask0  = ShfLAlnLSByte( mask0, 16 -index);
// convert upper 8 bytes of xmm: only least sig. 4 digits of output will be added to prev 16 digits
w1_u8 = _mm_cvtepi8_epi16(_mm_srli_si128 ( mask0, 8));
/* merge 2 digit  at a time with multiplier into each dword*/
w1_u8 = _mm_madd_epi16(w1_u8, _mm_loadu_si128(  (__m128i *) &MulplyQuadBaseExp3To0 [ 0]));
/* bits 63:0 has two dword integer, bits 63:32 is the LS dword of output; bits 127:64 is not needed*/
w1_u8 = _mm_cvtepu32_epi64( _mm_hadd_epi32(w1_u8, w1_u8) );
w3 = _mm_add_epi64(w3, _mm_srli_si128( w1_u8, 8) );
xxi = _mm_extract_epi64(w3, 0);
if( xxi >> 63 )
        return (NegSgn? (long long) RINT64VALNEG: (long long) RINT64VALPOS);
else   return (NegSgn? (long long) -xxi: (long long) xxi);

}
```

The general performance characteristics of an Intel SSE4.2-enhanced atol() replacement have a start-up cost that is somewhat lower than byte-granular implementations generated from C code.

Example 14-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing

```
// bit lookup table of valid ascii code for decimal string conversion, white space, sign, numeric digits
static  char BtMLValDecInt[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x28, 0xff, 0x03,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
```

```
// bit lookup table, white space only
static char BtMLws[32] = {0x0, 0x3e, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
// list of white space for sttni use
static  char listws[16] =
      {0x20, 0x9, 0xa, 0xb, 0xc, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
// list of numeric digits for sttni use
static  char rangenumint[16] =
      {0x30, 0x39, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
static  char rangenumintzr[16] =
      {0x30, 0x30, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};

// we use pmaddwd to merge two adjacent short integer pair, this is the second step of merging each pair of 2-
digit integers
static short MulplyPairBaseP2[8] =
{ 100, 1, 100, 1, 100, 1, 100, 1};

// Multiplier-pair for two adjacent short integer pair, this is the third step of merging each pair of 4-digit integers
static short MulplyPairBaseP4[8] =
{ 10000, 1, 10000, 1, 10000, 1, 10000, 1 };

// multiplier for pmulld for normalization of > 16 digits
static int MulplyByBaseExpN[8] =
{  10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000};

static short MulplyQuadBaseExp3To0[8] =
{ 1000, 100, 10, 1, 1000, 100, 10, 1};
__m128i __m128i_shift_right (__m128i value, int offset)
{ switch (offset)
   {
   case 1: value = _mm_srli_si128 (value, 1); break;
   case 2: value = _mm_srli_si128 (value, 2); break;
   case 3: value = _mm_srli_si128 (value, 3); break;
   case 4: value = _mm_srli_si128 (value, 4); break;
   case 5: value = _mm_srli_si128 (value, 5); break;
  case 6: value = _mm_srli_si128 (value, 6); break;
   case 7: value = _mm_srli_si128 (value, 7); break;
   case 8: value = _mm_srli_si128 (value, 8); break;
   case 9: value = _mm_srli_si128 (value, 9); break;
   case 10: value = _mm_srli_si128 (value, 10); break;
   case 11: value = _mm_srli_si128 (value, 11); break;
```

Example 14-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing (Contd.)

```
    case 12: value = _mm_srli_si128 (value, 12); break;
    case 13: value = _mm_srli_si128 (value, 13); break;
    case 14: value = _mm_srli_si128 (value, 14); break;
    case 15: value = _mm_srli_si128 (value, 15); break;
    }
  return value;
}

/* Load string at S near page boundary safely.  */

__m128i __m128i_strloadu_page_boundary (const  char *s)
{
  int offset = ((size_t) s & (16 - 1));
  if (offset)
  {
     __m128i v = _mm_load_si128 ((__m128i *) (s - offset));
     __m128i zero = _mm_setzero_si128 ();
     int bmsk = _mm_movemask_epi8 (_mm_cmpeq_epi8 (v, zero));
     if ( (bmsk >> offset) != 0 )  return __m128i_shift_right (v, offset);
  }
  return _mm_loadu_si128 ((__m128i *) s);
}
__m128i ShfLAlnLSByte( __m128i value, int offset)
{
   /*now remove constant bias, so each byte element are unsigned byte int */
   value = _mm_sub_epi8(value,  _mm_setr_epi32(0x30303030, 0x30303030, 0x30303030, 0x30303030));
   switch (offset)
   {
   case 1:
     value = _mm_slli_si128 (value, 1);   break;
   case 2:
     value = _mm_slli_si128 (value, 2);   break;
   case 3:
     value = _mm_slli_si128 (value, 3);   break;
   case 4:
     value = _mm_slli_si128 (value, 4);   break;
   case 5:
     value = _mm_slli_si128 (value, 5);   break;
   case 6:
     value = _mm_slli_si128 (value, 6);   break;
   case 7:
     value = _mm_slli_si128 (value, 7);   break;
   case 8:
     value = _mm_slli_si128 (value, 8);   break;
   case 9:
     value = _mm_slli_si128 (value, 9);   break;
   case 10:
     value = _mm_slli_si128 (value, 10);   break;
   case 11:
     value = _mm_slli_si128 (value, 11);   break;
   case 12:
     value = _mm_slli_si128 (value, 12);   break;
```

Example 14-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing (Contd.)

```
  case 13:
    value = _mm_slli_si128 (value, 13);   break;
  case 14:
    value = _mm_slli_si128 (value, 14);   break;
  case 15:
    value = _mm_slli_si128 (value, 15);   break;
  }
  return value;
}
```

With an input byte stream no more than 16 non-zero-leading digits, it has a constant performance. An input string consisting of more than 16 bytes of non-zero-leading digits can be processed in about 100 cycles or less, compared byte-granular solution needing around 200 cycles. Even for shorter input strings of 9 non-zero-leading digits, Intel SSE4.2 enhanced replacement can also achieve ~2X performance of byte-granular solutions.

14.5 NUMERICAL DATA CONVERSION TO ASCII FORMAT

Conversion of binary integer data to ASCII format gets used in many situations from simple C library functions to computations with finances. Some C libraries provides exported conversion functions like itoa, ltoa; other libraries implement internal equivalents to support data formatting needs of standard output functions. Among the most common binary integer to ascii conversion is conversion based on radix 10. Example 14-20 shows the basic technique implemented in many libraries for base 10 conversion to ascii of a 64-bit integer. For simplicity, the example produces lower-case output format.

Example 14-20. Conversion of 64-bit Integer to ASCII

```
// Convert 64-bit signed binary integer to lower-case ASCII format

static char lc_digits[]= "0123456789abcdefghijklmnopqrstuvwxyz";

int lltoa_cref( __int64 x,  char* out)
{const char *digits = &lc_digits[0];
char lbuf[32]  // base 10 conversion of 64-bit signed integer need only 21 digits
char * p_bkwd = &lbuf[2];
 __int64  y ;
unsigned int  base = 10, len = 0, r, cnt;
  if( x < 0)
  {   y = -x;
      while (y > 0)
      {   r = (int) (y % base); // one digit at a time from least significant digit
          y = y /base;
          * --p_bkwd = digits[r];
          len ++;
      }
      *out++ = '-';
      cnt = len +1;
      while( len--) *out++ = p_bkwd++; // copy each converted digits
  } else
```

Example 14-20. Conversion of 64-bit Integer to ASCII (Contd.)

```
  {
     y = x;
     while (y > 0)
     {   r = (int) (y % base); // one digit at a time from least significant digit
         y = y /base;
         * --p_bkwd = digits[r];
         len ++;
     }
     cnt = len;;
     while( len--) *out++ = p_bkwd++; // copy each converted digits
  }

  out[cnt] = 0;
  return (int) cnt;
}
```

Example 14-20 employs iterative sequence that process one digit at a time using the hardware native integer divide instruction. The reliance on integer divide can be replaced by fixed-point multiply technique discussed in Chapter 13, "64-bit Mode Coding Guidelines". This is shown in Example 14-21.

Example 14-21. Conversion of 64-bit Integer to ASCII without Integer Division

```
// Convert 64-bit signed binary integer to lower-case ASCII format and
// replace integer division with fixed-point multiply
;__int64 umul_64x64(__int64* p128, __int64 u, __int64 v)
umul_64x64 PROC
      mov     rax, rdx  ; 2nd parameter
      mul     r8  ; u * v
      mov     qword ptr [rcx], rax
      mov     qword ptr [rcx+8], rdx
      ret 0
umul_64x64 ENDP
#define   cg_10_pms3 0xccccccccccccccdull
static char lc_digits[]= "0123456789";

int lltoa_cref( __int64 x,  char* out)
{const char *digits = &lc_digits[0];
char lbuf[32]  // base 10 conversion of 64-bit signed integer need only 21 digits
char * p_bkwd = &lbuf[2];
 __int64  y, z128[2];
unsigned __int64   q;
unsigned int  base = 10, len = 0, r, cnt;
  if( x < 0)
  {   y = -x;
      while (y > 0)
      {  umul_64x64( &z128[0], y, cg_10_pms3);
         q = z128[1] >> 3;
         q = (y < q * (unsigned __int64) base)? q-1: q;
      r = (int) (y - q * (unsigned __int64) base); // one digit at a time from least significant digit
```

Example 14-21. Conversion of 64-bit Integer to ASCII without Integer Division (Contd.)

```
            y =q;
        * --p_bkwd = digits[r];
        len ++;
    }
    *out++ = '-';
    cnt = len +1;
    while( len--) *out++ = p_bkwd++; // copy each converted digits
  } else

  {
    y = x;
    while (y > 0)
    {   umul_64x64( &z128[0], y, cg_10_pms3);
        q = z128[1] >> 3;
        q = (y < q * (unsigned __int64) base)? q-1: q;
        r = (int) (y - q * (unsigned __int64) base); // one digit at a time from least significant digit
        y =q;
        * --p_bkwd = digits[r];
        len ++;
    }
    cnt = len;;
    while( len--) *out++ = p_bkwd++; // copy each converted digits
  }
  out[cnt] = 0;
  return cnt;
}
```

Example 14-21 provides significant speed improvement by eliminating the reliance on integer divisions. However, the numeric format conversion problem is still constrained by the dependent chain that process one digit at a time.

SIMD technique can apply to this class of integer numeric conversion problem by noting that an unsigned 64-bit integer can expand a dynamic range of up to twenty digits. Such a wide dynamic range can be expressed as polynomial expressions of the form:

$$a_0 + a_1 * 10^4 + a_2 * 10^8 + a_3 * 10^{12} + a_4 * 10^{16} \text{ where}$$

the dynamic range of a_i is between [0, 9999].

Reduction of an unsigned 64-bit integer into up-to 5 reduced-range coefficients can be computed using fixed-point multiply in stages. Once the dynamic range of coefficients are reduced to no more than 4 digits, one can apply SIMD techniques to compute ascii conversion in parallel.

The SIMD technique to convert an unsigned 16-bit integer via radix 10 with input dynamic range [0, 9999] is shown in Figure 14-4. This technique can also be generalized to apply to other non-power-of-2 radix that is less than 16.

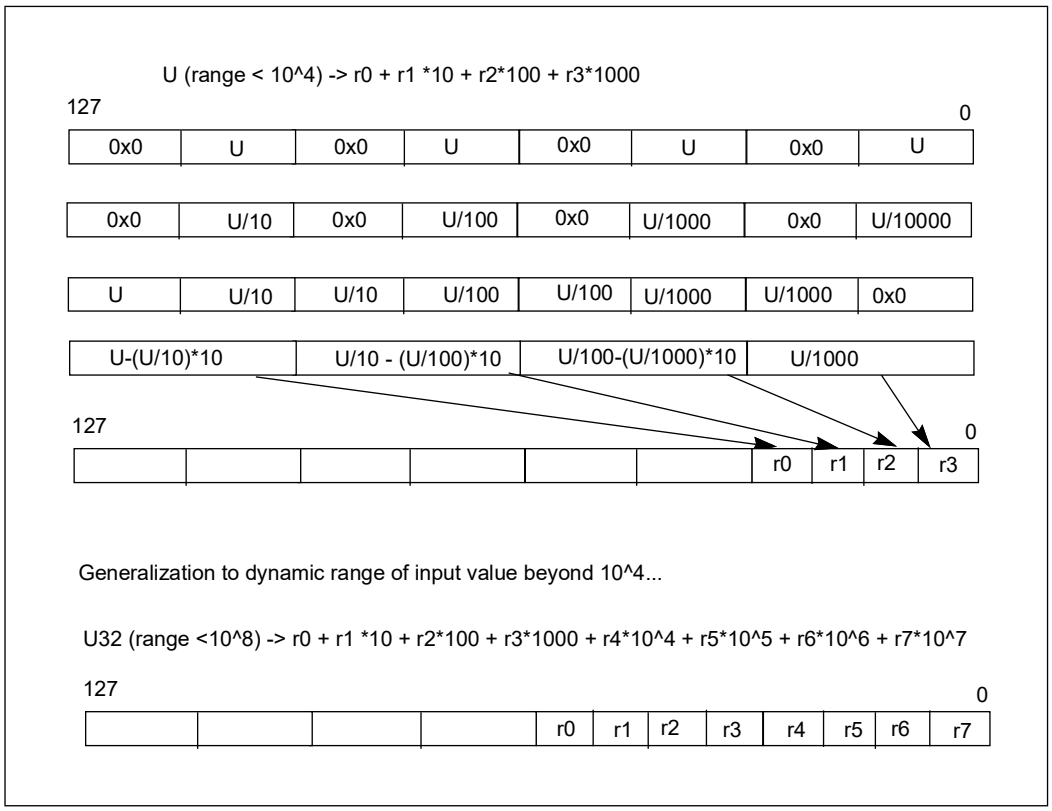


Figure 14-4. Compute Four Remainders of Unsigned Short Integer in Parallel

To handle greater input dynamic ranges, the input is reduced into multiple unsigned short integers and converted sequentially. The most significant U16 conversion is computed first, followed by the conversion of the next four significant digits.

Example 14-22 shows the fixed-point multiply combined with parallel remainder computation using SSE4 instructions for 64-bit integer conversion up to 19 digits.

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4

```
#include <smmintrin.h>
#include <stdio.h>
#define   QWCG10to8        0xabcc77118461cefdull
#define   QWCONST10to8   100000000ull

/* macro to convert input parameter of short integer "hi4" into output variable "x3" which is __m128i;
  the input value "hi4" is assume to be less than 10^4;
  the output is 4  single-digit integer between 0-9, located in the low byte of each dword,
  most significant digit in lowest DW.
  implicit overwrites: locally allocated __m128i variable "x0", "x2"
*/
#define   __ParMod10to4SSSE3( x3, hi4 ) \
{                                       \
      x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128(  (hi4)), 0); \
      x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
      x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d)), 10); \
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
      (x3) = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) (hi4), 1);  \
      (x3) = _mm_or_si128(x2, (x3));\
      (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}

/* macro to convert input parameter of the 3rd dword element of "t5" ( __m128i type)
  into output variable "x3" which is __m128i;
  the third dword element "t5" is assume to be less than 10^4, the 4th dword must be 0;
  the output is 4  single-digit integer between 0-9, located in the low byte of each dword,
  MS digit in LS DW.
  implicit overwrites: locally allocated __m128i variable "x0", "x2"
*/

#define   __ParMod10to4SSSE3v( x3, t5 ) \
{                                       \
      x0 = _mm_shuffle_epi32( t5, 0xaa);  \
      x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
      x2 = _mm_srli_epi32( _mm_madd_epi16( x2, _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d)), 10);
\
      (x3) = _mm_or_si128(_mm_slli_si128(x2, 6),   _mm_srli_si128(t5, 6));  \
      (x3) = _mm_or_si128(x2, (x3));\
      (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}
static __attribute__ ((aligned(16)) ) short  quo4digComp_mulplr_d[8] =
{  1024, 0, 64, 0, 8, 0, 0, 0};
static __attribute__ ((aligned(16)) )  short  quoTenThsn_mulplr_d[8] =
{  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static __attribute__ ((aligned(16)) )  short  mten_mulplr_d[8] =
{ -10, 1, -10,  1, -10,  1, -10, 1};
static __attribute__ ((aligned(16)) )  unsigned short bcstpklodw[8] =
{0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw1[8] =
{0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw2[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004, 0x8080, 0x8080};
static __attribute__ ((aligned(16)) )  unsigned short bcstpkdw3[8] =
{0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x8080, 0x080c, 0x0004};
static __attribute__ ((aligned(16)) )  int  asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};
static __attribute__ ((aligned(16)) )  int  asc0reversebias[4] =
{0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0, 0xd0d0d0d0};
static __attribute__ ((aligned(16)) )  int   pr_cg_10to4[4] =
{  0x68db8db, 0 , 0x68db8db, 0};
static __attribute__ ((aligned(16)) )  int   pr_1_m10to4[4] =
{  -10000, 0 , 1, 0};
/*input value "xx" is less than 2^63-1 */
/* In environment that does not support binary integer arithmetic on __int128_t,
   this helper can be done as asm routine
*/
__inline __int64_t u64mod10to8( __int64_t * pLo, __int64_t xx)
{__int128_t t, b = (__int128_t)QWCG10to8;
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
__int64_t q;
  t = b * (__int128_t)xx;
  q = t>>(64 +26); // shift count associated with QWCG10to8
  *pLo = xx - QWCONST10to8 * q;
  return q;
}
/* convert integer between 2^63-1 and 0 to ASCII string */
int  sse4i_q2a_u63 ( __int64_t xx, char *ps)
{int  j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, m1;
long long  w, u;
  if ( xx < 10000 )
  {   j = ubs_Lt10k_2s_i2 ( (unsigned ) xx, ps);
      ps[j] = 0;       return j;
  }
  if (xx < 100000000 )   // dynamic range of xx is less than 32-bits
  {   m0 = _mm_cvtsi32_si128( xx);
      x1 = _mm_shuffle_epi32(m0, 0x44);  // broadcast to dw0 and dw2
      x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // quotient in dw2, remainder in dw0
      __ParMod10to4SSSE3v( x3, m0); // pack single digit from each dword to dw0
      x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
      __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));   // move the remainder to dw2 first
      x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
      x4 = _mm_or_si128(x4, x5); // pack  digits in bytes 0-7 with leading 0
      cnt = 8;
  }
  else
  {   hi8 = u64mod10to8(&lo8, xx);
      if ( hi8 < 10000) // decompose lo8 dword into quotient and remainder mod 10^4
      {   m0 = _mm_cvtsi32_si128( lo8);
          x2 = _mm_shuffle_epi32(m0, 0x44);
          x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
          x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
          m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);  // quotient in dw0
          __ParMod10to4SSSE3( x3, hi8); // handle digist 11:8 first
          x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
          __ParMod10to4SSSE3v( x3, m0);  // handle digits 7:4
          x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
          x4 = _mm_or_si128(x4, x5);
          __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
          x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
          x4 = _mm_or_si128(x4, x5); // pack single digist in bytes 0-11 with leading 0
          cnt = 12;
      }
      else
      {   cnt = 0;
          if ( hi8 >=  100000000)  // handle input greater than 10^16
          {   abv16 = u64mod10to8(&temp, (__int64_t)hi8);
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
        hi8 = temp;
        __ParMod10to4SSSE3( x3, abv16);
        x6 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
        cnt = 4;
        } // start with handling digits 15:12
        m0 = _mm_cvtsi32_si128( hi8);
        x2 = _mm_shuffle_epi32(m0, 0x44);
        x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
        x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
        m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
        m1 = _mm_cvtsi32_si128( lo8);
        x2 = _mm_shuffle_epi32(m1, 0x44);
        x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
        x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
        m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
        __ParMod10to4SSSE3v( x3, m0);
        x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
        __ParMod10to4SSSE3v( x3, _mm_slli_si128(m0, 8));
        x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw1) );
        x4 = _mm_or_si128(x4, x5);
        __ParMod10to4SSSE3v( x3, m1);
        x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw2) );
        x4 = _mm_or_si128(x4, x5);
        __ParMod10to4SSSE3v( x3, _mm_slli_si128(m1, 8));
        x5 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpkdw3) );
        x4 = _mm_or_si128(x4, x5);
        cnt += 16;
     }
  }
  m0 = _mm_loadu_si128( (__m128i *) asc0reversebias);
  if( cnt > 16)
  {  tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x6,_mm_setzero_si128()) );
     x6 = _mm_sub_epi8(x6, m0);
  } else {
     tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(x4,_mm_setzero_si128()) );
  }
#ifndef __USE_GCC__
  __asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
  _BitScanForward(&idx, tmp);
#endif
  x4 = _mm_sub_epi8(x4, m0);
  cnt  -= idx;
  w = _mm_cvtsi128_si64(x4);

  switch(cnt)
  {  case5:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
        break;
     case6:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
        break;
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
      case7:*ps = (char) (w >>8);   *(short *) (&ps[1]) = (short) (w >>16);
         *(unsigned *) (&ps[3]) = (w >>32);
         break;
      case 8: *(long long *)ps = w;
      break;
      case9:*ps++ = (char) (w >>24);
         *(long long *) (&ps[0]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
         break;
      case10:*(short *)ps = (short) (w >>16);
         *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
         break;
      case11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
            *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
         break;
      case 12: *(unsigned *)ps = w;
         *(long long *) (&ps[4])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 4));
         break;
      case13:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
         *(long long *) (&ps[4]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
         break;
      case14:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
         *(long long *) (&ps[6]) =  _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
      break;
      case15: *ps = (char) (w >>8);
         *(short *) (&ps[1]) = (short) (w >>16);     *(unsigned *) (&ps[3]) = (w >>32);
         *(long long *) (&ps[7])  = _mm_cvtsi128_si64( _mm_srli_si128(x4, 8));
         break;
      case 16: _mm_storeu_si128( (__m128i *) ps, x4);
         break;
      case17:u = _mm_cvtsi128_si64(x6);  *ps++ = (char) (u >>24);
         _mm_storeu_si128( (__m128i *) &ps[0], x4);
         break;
      case18:u = _mm_cvtsi128_si64(x6);  *(short *)ps = (short) (u >>16);
         _mm_storeu_si128( (__m128i *) &ps[2], x4);
         break;
      case19:u = _mm_cvtsi128_si64(x6);  *ps = (char) (u >>8);
         *(short *) (&ps[1]) = (short) (u >>16);
         _mm_storeu_si128( (__m128i *) &ps[3], x4);
         break;
      case20:u = _mm_cvtsi128_si64(x6);  *(unsigned *)ps = (short) (u);
         _mm_storeu_si128( (__m128i *) &ps[4], x4);
         break;
   }
   return cnt;
}
/* convert input value into 4 single digits via parallel fixed-point arithmetic with each dword
   element, and pack each digit into low dword element and write to buffer without leading
   white space; input value must be < 10000 and > 9
*/
__inline  int ubs_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
__m128i x0, m0, x2, x3, x4,  compv;
// Use a set of scaling constant to compensate for lack for per-element shift count
  compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
// broadcast input value to each dword element
  x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128(  x_Lt10k), 0);
  // low to high dword in x0 : u16, u16, u16, u16
  m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d); // load 4 congruent consts
  x2 = _mm_mulhi_epu16(x0, m0); // parallel fixed-point multiply for base 10,100, 1000, 10000
  x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10);
// dword content in x2: u16/10, u16/100, u16/1000, u16/10000
  x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
  //word content in x3:  0, u16, 0, u16/10, 0, u16/100, 0, u16/1000

  x4 = _mm_or_si128(x2, x3);
  // perform parallel remainder operation with each word pair to derive 4 unbiased single-digit result
  x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
  x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
  // pack each ascii-biased digits from respective dword to the low dword element
  x3 = _mm_shuffle_epi8(x2, _mm_loadu_si128( (__m128i *) bcstpklodw) );
```

```
  // store ascii result to buffer without leading white space
  if (x_Lt10k > 999 )
  {   *(int *) ps = _mm_cvtsi128_si32(  x3);
      return 4;
  }
  else if (x_Lt10k > 99 )
  {   tmp = _mm_cvtsi128_si32(  x3);
      *ps = (char ) (tmp >>8);
      *((short *) (++ps)) = (short ) (tmp >>16);
      return 3;
  }
  else if ( x_Lt10k > 9) // take advantage of reduced dynamic range > 9 to reduce branching
  {   *((short *) ps) = (short ) _mm_extract_epi16( x3, 1);
      return 2;
  }
  *ps = '0' + x_Lt10k;
  return 1;
}
char lower_digits[] = "0123456789";

int ltoa_sse4 (const long long s1, char * buf)
{long long temp ;
int j = 1, len = 0;
const char *digits = &lower_digits[0];
  if( s1 < 0) {

      temp = -s1;
      len ++;
       beg[0] = '-';
      if( temp < 10) beg[1] = digits[ (int) temp];
      else len += sse4i_q2a_u63( temp, &buf[ 1]); // parallel conversion in 4-digit granular operation
  }
```

Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 (Contd.)

```
  else {
  if( s1 < 10) beg[ 0 ] = digits[(int)s1];
  else len += sse4i_q2a_u63( s1, &buf[ 1] );
  }
  buf[len] = 0;
  return len;
}
```

When an ltoa()-like utility implementation executes native IDIV instruction to convert one digit at a time, it can produce output at a speed of about 45-50 cycles per digit. Using fixed-point multiply to replace IDIV (like Example 14-21) can reduce 10-15 cycles per digit.

The range-reduction technique demonstrated in Example 14-22 reduces up-to 19 levels of dependency chain down to a five-level hierarchy and allows parallel SIMD technique to perform 4-wide numeric conversion. This technique can also be done with only Intel SSSE3, and with similar speed improvement.

Support for conversion to wide character strings can be easily adapted using the code snippet shown in Example 14-23.

Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4

```
static __attribute__ ((aligned(16)) )  int  asc0bias[4] =
{0x30, 0x30, 0x30, 0x30};

// exponent_x must be < 10000 and > 9
__inline  int ubs_Lt10k_2wcs_i2(int x_Lt10k, wchar_t *ps)
{
__m128i x0, m0, x2, x3, x4,  compv;
  compv = _mm_loadu_si128( (__m128i *) quo4digComp_mulplr_d);
  x0 = _mm_shuffle_epi32( _mm_cvtsi32_si128(  x_Lt10k), 0);  // low to high dw: u16, u16, u16, u16
  m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d);
  // u16, 0, u16, 0, u16, 0, u16, 0
  x2 = _mm_mulhi_epu16(x0, m0);
  x2 = _mm_srli_epi32( _mm_madd_epi16( x2, compv), 10); // u16/10, u16/100, u16/1000, u16/10000

  x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);  // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
  x4 = _mm_or_si128(x2, x3);
  x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;
     x2 = _mm_add_epi32( x4, _mm_loadu_si128( (__m128i *) asc0bias) ) ;
  x2 = _mm_shuffle_epi32(x2, 0x1b);  // switch sequence
  if (x_Lt10k > 999 ) {
     _mm_storeu_si128( (__m128i *) ps, x2);
     return 4;
  }
  else if (x_Lt10k > 99 ) {
     *ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x2, 4));
     *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
     return 3;
  }
```

Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 (Contd.)

```
   else if ( x_Lt10k > 9){ // take advantage of reduced dynamic range > 9 to reduce branching
      *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x2, 8));
      return 2;
   }
   *ps = L'0' + x_Lt10k;
   return 1;
}

long long  sse4i_q2wcs_u63 ( __int64_t xx, wchar_t *ps)
{int  j, tmp, idx=0, cnt;
__int64_t lo8, hi8, abv16, temp;
__m128i x0, m0, x1, x2, x3, x4, x5, x6, x7, m1;

   if ( xx < 10000 ) {
      j = ubs_Lt10k_2wcs_i2 ( (unsigned ) xx, ps); ps[j] = 0; return j;
   }
   if (xx < 100000000 ) {  // dynamic range of xx is less than 32-bits
      m0 = _mm_cvtsi32_si128( xx);
      x1 = _mm_shuffle_epi32(m0, 0x44);  // broadcast to dw0 and dw2
      x3 = _mm_mul_epu32(x1, _mm_loadu_si128( (__m128i *) pr_cg_10to4 ));
      x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
      m0 = _mm_add_epi32( _mm_srli_si128( x1, 8), x3); // quotient in dw2, remainder in dw0
      __ParMod10to4SSSE3v( x3, m0);
      //x4 = _mm_shuffle_epi8(x3, _mm_loadu_si128( (__m128i *) bcstpklodw) );
      x3 = _mm_shuffle_epi32(x3, 0x1b);
      __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8));   // move the remainder to dw2 first
      x4 = _mm_shuffle_epi32(x4, 0x1b);
      cnt = 8;
   } else {
      hi8 = u64mod10to8(&lo8, xx);
      if( hi8 < 10000) {
       m0 = _mm_cvtsi32_si128( lo8);
       x2 = _mm_shuffle_epi32(m0, 0x44);
       x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
       x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));

       m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
       __ParMod10to4SSSE3( x3, hi8);
       x3 = _mm_shuffle_epi32(x3, 0x1b);
       __ParMod10to4SSSE3v( x4, m0);
       x4 = _mm_shuffle_epi32(x4, 0x1b);
__ParMod10to4SSSE3v( x5, _mm_slli_si128(m0, 8));
       x5 = _mm_shuffle_epi32(x5, 0x1b);
       cnt = 12;
      } else {
       cnt = 0;
       if ( hi8 >  100000000) {
         abv16 = u64mod10to8(&temp, (__int64_t)hi8);
```

Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 (Contd.)

```
 hi8 = temp;
        __ParMod10to4SSSE3( x7, abv16);
        x7 = _mm_shuffle_epi32(x7, 0x1b);
        cnt = 4;
       }
       m0 = _mm_cvtsi32_si128( hi8);
       x2 = _mm_shuffle_epi32(m0, 0x44);
       x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
       x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
       m0 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
       m1 = _mm_cvtsi32_si128( lo8);
       x2 = _mm_shuffle_epi32(m1, 0x44);
       x3 = _mm_mul_epu32(x2, _mm_loadu_si128( (__m128i *)pr_cg_10to4));
       x3 = _mm_mullo_epi32(_mm_srli_epi64(x3, 40), _mm_loadu_si128( (__m128i *)pr_1_m10to4));
       m1 = _mm_add_epi32( _mm_srli_si128( x2, 8), x3);
       __ParMod10to4SSSE3v( x3, m0);
       x3 = _mm_shuffle_epi32(x3, 0x1b);
       __ParMod10to4SSSE3v( x4, _mm_slli_si128(m0, 8));
       x4 = _mm_shuffle_epi32(x4, 0x1b);
       __ParMod10to4SSSE3v( x5, m1);
       x5 = _mm_shuffle_epi32(x5, 0x1b);
       __ParMod10to4SSSE3v( x6, _mm_slli_si128(m1, 8));
       x6 = _mm_shuffle_epi32(x6, 0x1b);
       cnt += 16;
     }
   }

   m0 = _mm_loadu_si128( (__m128i *) asc0bias);
   if( cnt > 16) {
      tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x7,_mm_setzero_si128()) ) ;
      //x7 = _mm_add_epi32(x7, m0);
   } else {
      tmp = _mm_movemask_epi8( _mm_cmpgt_epi32(x3,_mm_setzero_si128()) );
   }
#ifndef __USE_GCC__
   __asm__ ("bsfl %1, %%ecx; movl %%ecx, %0;" :"=r"(idx) :"r"(tmp) : "%ecx");
#else
   _BitScanForward(&idx, tmp);
#endif
   x3 = _mm_add_epi32(x3, m0);
   cnt  -= (idx >>2);
   x4 = _mm_add_epi32(x4, m0);
   switch(cnt) {
   case5:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
        _mm_storeu_si128( (__m128i *) ps, x4);
   break;
   case6:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
        _mm_storeu_si128( (__m128i *) &ps[2], x4);
   break;
case7:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
     *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
     _mm_storeu_si128( (__m128i *) &ps[2], x4);
```

Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 (Contd.)

```
      break;
   case 8: _mm_storeu_si128( (__m128i *) &ps[0], x3);
      _mm_storeu_si128( (__m128i *) &ps[4], x4);
   break;
   case9:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) ps, x4);
      _mm_storeu_si128( (__m128i *) &ps[4], x5);
   break;
   case10:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[2], x4);
      _mm_storeu_si128( (__m128i *) &ps[6], x5);
   break;

   case11:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
      *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[2], x4);
      _mm_storeu_si128( (__m128i *) &ps[6], x5);
   break;
   case 12: _mm_storeu_si128( (__m128i *) &ps[0], x3);
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[4], x4);
      _mm_storeu_si128( (__m128i *) &ps[8], x5);
   break;
   case13:*ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 12));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) ps, x4);
      x6 = _mm_add_epi32(x6, m0);
      _mm_storeu_si128( (__m128i *) &ps[4], x5);
      _mm_storeu_si128( (__m128i *) &ps[8], x6);
   break;
   case14:*(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[2], x4);
      x6 = _mm_add_epi32(x6, m0);
      _mm_storeu_si128( (__m128i *) &ps[6], x5);
      _mm_storeu_si128( (__m128i *) &ps[10], x6);
   break;
   case15:*ps++ =  (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x3, 4));
      *(long long *) ps = _mm_cvtsi128_si64( _mm_srli_si128( x3, 8));
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[2], x4);
      x6 = _mm_add_epi32(x6, m0);
      _mm_storeu_si128( (__m128i *) &ps[6], x5);
      _mm_storeu_si128( (__m128i *) &ps[10], x6);
   break;
case 16: _mm_storeu_si128( (__m128i *) &ps[0], x3);
      x5 = _mm_add_epi32(x5, m0);
      _mm_storeu_si128( (__m128i *) &ps[4], x4);
```

Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 (Contd.)

```
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[8], x5);
        _mm_storeu_si128( (__m128i *) &ps[12], x6);
    break;

    case17:x7 = _mm_add_epi32(x7, m0);
        *ps++ = (wchar_t) _mm_cvtsi128_si32( _mm_srli_si128( x7, 12));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) ps, x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[4], x4);
        _mm_storeu_si128( (__m128i *) &ps[8], x5);
        _mm_storeu_si128( (__m128i *) &ps[12], x6);
    break;
    case18:x7 = _mm_add_epi32(x7, m0);
        *(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[2], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[6], x4);
        _mm_storeu_si128( (__m128i *) &ps[10], x5);
        _mm_storeu_si128( (__m128i *) &ps[14], x6);
    break;
    case19:x7 = _mm_add_epi32(x7, m0);
        *ps++ = (wchar_t) _mm_cvtsi128_si64( _mm_srli_si128( x7, 4));
        *(long long *)ps =  _mm_cvtsi128_si64( _mm_srli_si128( x7, 8));
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[2], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[6], x4);
        _mm_storeu_si128( (__m128i *) &ps[10], x5);
        _mm_storeu_si128( (__m128i *) &ps[14], x6);
    break;
    case20:x7 = _mm_add_epi32(x7, m0);
        _mm_storeu_si128( (__m128i *) &ps[0], x7);
        x5 = _mm_add_epi32(x5, m0);
        _mm_storeu_si128( (__m128i *) &ps[4], x3);
        x6 = _mm_add_epi32(x6, m0);
        _mm_storeu_si128( (__m128i *) &ps[8], x4);
        _mm_storeu_si128( (__m128i *) &ps[12], x5);
        _mm_storeu_si128( (__m128i *) &ps[16], x6);
    break;
    }
    return cnt;
}
```

14.5.1 LARGE INTEGER NUMERIC COMPUTATION

14.5.1.1 MULX Instruction and Large Integer Numeric Computation

The MULX instruction is similar to the MUL instruction but does not read or write arithmetic flags and is enhanced with more flexibility in register allocations for the destination operands. These enhancements allow better out-of-order operation of the hardware and for software to intermix add-carry instruction without corrupting the carry chain.

For computations calculating large integers (e.g. 2048-bit RSA key), MULX can improve performance significantly over techniques based on MUL/ADC chain sequences. Intel AVX2 can be used to build efficient techniques, see Section 15.6.2.

Example 14-24 gives an example of how MULX is used to improve the carry chain computation of integer numeric greater than 64-bit wide.

Example 14-24. MULX and Carry Chain in Large Integer Numeric

```
mov rax, [rsi+8*1]
mul rbp ; rdx:rax = rax * rbp
mov r8, rdx
add r9, rax
adc r8, 0
add r9, rbx
adc r8, 0
```

```
mulx rbx, r8, [rsi+8*1] ; rbx:r8 = rdx * [rsi+8*1]
add r8, r9
adc rbx, 0
add r8, rbp
adc rbx, 0
```

Using MULX to implement 128-bit integer output can be a useful building block for implementing library functions ranging from atof/strtod or intermediate mantisa computation or mantissa/exponent normalization in 128-bit binary decimal floating-point operations. Example 14-25 gives examples of building-block macros, used in 128-bit binary-decimal floating-point operations, which can take advantage MULX to calculate intermediate results of multiple-precision integers of widths between 128 to 256 bits. Details of binary-integer-decimal (BID) floating-point format and library implementation of BID operation can be found at the Intel® Decimal Floating-Point Math Library.

Example 14-25. Building-block Macro Used in Binary Decimal Floating-point Operations

```
// Portable C macro of 64x64-bit product using 32-bit word granular operations
// Output: BID_UINT128 P128
#define __mul_64x64_to_128MACH(P128, CX64, CY64)  \
{                                     \
  BID_UINT64 CXH,CXL,CYH,CYL,PL,PH,PM,PM2;    \
  CXH = (CX64) >> 32;                 \
  CXL = (BID_UINT32)(CX64);                 \
  CYH = (CY64) >> 32;                 \
  CYL = (BID_UINT32)(CY64);                 \
  PM = CXH*CYL;                       \
  PH = CXH*CYH;                       \
  PL = CXL*CYL;                      \
  PM2 = CXL*CYH;                      \
  PH += (PM>>32);                     \
  PM = (BID_UINT64)((BID_UINT32)PM)+PM2+(PL>>32);       \
  (P128).w[1] = PH + (PM>>32);              \
  (P128).w[0] = (PM<<32)+(BID_UINT32)PL;          \
}
```

Example 14-25. Building-block Macro Used in Binary Decimal Floating-point Operations (Contd.)

```
// 64x64-bit product using intrinsic producing 128-bit output in 64-bit mode
// Output: BID_UINT128 P128
#define __mul_64x64_to_128MACH_x64(P128, CX64, CY64)  \
{                                     \
(P128).w[0] = mulx_u64(CX64, CY64, &( (P128).w[1]) );             \
}
```

CHAPTER 15
OPTIMIZATIONS FOR INTEL® AVX, INTEL® AVX2, & INTEL® FMA

Intel® Advanced Vector Extension (Intel® AVX), is a major enhancement to Intel Architecture. It extends the functionality of previous generations of 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) vector instructions and increased the vector register width to support 256-bit operations. The Intel AVX ISA enhancement is focused on float-point instructions. Some 256-bit integer vectors are supported via floating-point to integer and integer to floating-point conversions.

The Sandy Bridge microarchitecture implements the Intel AVX instructions, in most cases, on 256-bit hardware. Thus, each core has 256-bit floating-point Add and Multiply units. The Divide and Square-root units are not enhanced to 256-bits. Thus, Intel AVX instructions use the 128-bit hardware in two steps to complete these 256-bit operations.

Prior generations of Intel® SSE instructions generally are two-operand syntax, where one of the operands serves both as source and as destination. Intel AVX instructions are encoded with a VEX prefix, which includes a bit field to encode vector lengths and support three-operand syntax. A typical instruction has two sources and one destination. Four operand instructions such as VBLENDVPS and VBLENDVPD exist as well. The added operand enables non-destructive source (NDS) and it eliminates the need for register duplication using MOVAPS operations.

With the exception of MMX™ instructions, almost all legacy 128-bit Intel SSE instructions have Intel AVX equivalents that support three operand syntax. 256-bit Intel AVX instructions employ three-operand syntax and some with 4-operand syntax.

The 256-bit vector register **YMM** extends the 128-bit **XMM** register to 256 bits. Thus the lower 128-bits of YMM is aliased to the legacy XMM registers.

While 256-bit Intel AVX instructions writes 256 bits of results to YMM, 128-bit Intel AVX instructions writes 128-bits of results into the XMM register and zeros the upper bits above bit 128 of the corresponding YMM. 16 vector registers are available in 64-bit mode. Only the lower 8 vector registers are available in non-64-bit modes.

Software can continue to use any mixture of legacy Intel SSE code, 128-bit Intel AVX code and 256-bit Intel AVX code. Section covers guidelines to deliver optimal performance across mixed-vector-length code modules without experiencing transition delays between legacy Intel SSE and Intel AVX code. There are no transition delays of mixing 128-bit Intel AVX code and 256-bit Intel AVX code.

The optimal memory alignment of an Intel AVX 256-bit vector, stored in memory, is 32 bytes. Some data-movement 256-bit Intel AVX instructions enforce 32-byte alignment and will signal #GP fault if memory operand is not properly aligned. Most 256-bit Intel AVX instructions do not require address alignment. These instructions generally combine load and compute operations, so any non-aligned memory address can be used in these instructions.

For best performance, software should pay attention to align the load and store addresses to 32 bytes whenever possible. Most examples can be found in the Chapter 15 GitHub Repository. A link to each example there has been added to its corresponding example in this chapter.

The major differences between using Intel AVX instructions and legacy Intel SSE instructions are summarized in Table 15-1.

Table 15-1. Features Between 256-bit Intel® AVX, 128-bit Intel® AVX, and Legacy Intel® SSE Extensions

Features	256-bit AVX	128-bit AVX	Legacy SSE-AESNI
Functionality Scope	Floating-point operation, Data Movement.	Matches legacy SIMD ISA (except MMX).	128-bit FP and integer SIMD ISA.
Register Operand	YMM.	XMM.	XMM.
Operand Syntax	Up to 4; non-destructive source.	Up to 4; non-destructive source.	2 operand syntax; destructive source.
Memory alignment	Load-Op semantics do not require alignment.	Load-Op semantics do not require alignment.	Always enforce 16B alignment.
Aligned Move Instructions	32 byte alignment.	16 byte alignment.	16 byte alignment.
Non-destructive source operand	Yes	Yes	No
Register State Handling	Updates bits 255:0.	Updates 127:0; Zeros bits above 128.	Updates 127:0; Bits above 128 unmodified.
Intrinsic Support	New 256-bit data types. _mm256 prefix for promoted functionality. New intrinsics for new functionalities.	Existing data types. Inherit same prototype for exiting functionalities. Use “_mm” prefix for new VEX-128 functionalities.	Baseline datatypes and prototype definitions.
128-bit Lanes	Applies to most 256-bit operations.	One 128-bit lane.	One 128-bit lane.
Mixed Code Handling	Use VZEROUPPER to avoid transition penalty.	No transition penalty.	Transition penalty after executing 256-bit AVX code.

15.1 INTEL® AVX INTRINSICS CODING

256-bit Intel AVX instructions have new intrinsics. Specifically, 256-bit Intel AVX instruction that are promoted to 256-bit vector length from existing Intel SSE functionality are generally prototyped with a “_mm256” prefix instead of the “_mm” prefix and using new data types defined for 256-bit operation. New functionality in 256-bit AVX instructions have brand new prototype.

The 128-bit Intel AVX instruction that were promoted from legacy SIMD ISA uses the same prototype as before. Newer functionality common in 256-bit and 128-bit AVX instructions are prototyped with “_mm256” and “_mm” prefixes respectively.

Thus porting from legacy SIMD code written in intrinsic can be ported to 256-bit Intel AVX code with a modest effort.

The following guidelines show how to convert a simple intrinsic from Intel SSE code sequence to Intel AVX:

- Align statically and dynamically allocated buffers to 32-bytes.
- May need to double supplemental buffer size.
- Change __mm_ intrinsic name prefix with __mm256_.
- Change variable data types names from __m128 to __m256.
- Divide by 2 iteration count (or double stride length).

The example below[1] on Cartesian coordinate transformation demonstrates the Intel AVX Instruction format, 32 byte YMM registers, dynamic and static memory allocation with data alignment of 32bytes, and the C data type representing 8 floating-point elements in a YMM register.

Example 15-1. Cartesian Coordinate Transformation with Intrinsics

```
//Use SSE intrinsic
#include "wmmintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 16);
  float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 16);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
//Static memory allocation of 4 floats with 16byte
alignment
__declspec(align(16)) float cos_sin_theta_vec[4] =
{cos_theta, sin_theta, cos_theta, sin_theta};

__declspec(align(16)) float sin_cos_theta_vec[4] =
{sin_theta, cos_theta, sin_theta, cos_theta};

//__m128 data type represents an xmm
  //register with 4 float elements
  __m128 Xmm_cos_sin =
_mm_load_ps(cos_sin_theta_vec);
```

```
// Use Intel AVX intrinsic
#include "immintrin.h"

int main()
{ int len = 3200;
  //Dynamic memory allocation with 32byte
  //alignment
  float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 32);
  float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 32);
//init data
for(int i=0; i<len; i++) pInVector[i] = 1;

float cos_theta = 0.8660254037;
  float sin_theta = 0.5;
//Static memory allocation of 8 floats with 32byte
alignment
__declspec(align(32)) float cos_sin_theta_vec[8] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};

__declspec(align(32)) float sin_cos_theta_vec[8] =
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta };

  //__m256 data type holds 8 float elements
  __m256 Ymm_cos_sin = _mm256_-
load_ps(cos_sin_theta_vec);
```

1. See Example 1 in GitHub.

Example 15-1. Cartesian Coordinate Transformation with Intrinsics (Contd.)

```
//SSE 128bit packed single load
  __m128 Xmm_sin_cos =
_mm_load_ps(sin_cos_theta_vec);

__m128 Xmm0, Xmm1, Xmm2, Xmm3;

//processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=8)
 {
   Xmm0 = _mm_load_ps(pInVector+i);
   Xmm1 = _mm_moveldup_ps(Xmm0);
   Xmm2 = _mm_movehdup_ps(Xmm0);
   Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
   Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
   Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
   _mm_store_ps(pOutVector + i, Xmm3);

 Xmm0 = _mm_load_ps(pInVector+i+4);
   Xmm1 = _mm_moveldup_ps(Xmm0);
   Xmm2 = _mm_movehdup_ps(Xmm0);
   Xmm1 = _mm_mul_ps(Xmm1,Xmm_cos_sin);
   Xmm2 = _mm_mul_ps(Xmm2,Xmm_sin_cos);
   Xmm3 = _mm_addsub_ps(Xmm1, Xmm2);
   _mm_store_ps(pOutVector+i+4, Xmm3);
 }
_mm_free(pInVector);
  _mm_free(pOutVector);
return 0;
}
```

```
 //AVX 256bit packed single load
  __m256 Ymm_sin_cos = _mm256_-
load_ps(sin_cos_theta_vec);
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
//processing 8 elements in an unrolled twice loop

for(int i=0; i<len; i+=16)
 {
   Ymm0 = _mm256_load_ps(pInVector+i);
   Ymm1 = _mm256_moveldup_ps(Ymm0);
   Ymm2 = _mm256_movehdup_ps(Ymm0);
   Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
   Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
   Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
   _mm256_store_ps(pOutVector + i, Ymm3);


 Ymm0 = _mm256_load_ps(pInVector+i+8);
   Ymm1 = _mm256_moveldup_ps(Ymm0);
   Ymm2 = _mm256_movehdup_ps(Ymm0);
   Ymm1 = _mm256_mul_ps(Ymm1,Ymm_cos_sin);
   Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
   Ymm3 = _mm256_addsub_ps(Ymm1, Ymm2);
   _mm256_store_ps(pOutVector+i+8, Ymm3);
 }
_mm_free(pInVector);
  _mm_free(pOutVector);
return 0;
}
```

15.1.1 INTEL® AVX ASSEMBLY CODING

Similar to the intrinsic porting guidelines, assembly porting guidelines include:

- Align statically and dynamically allocated buffers to 32-bytes.
- Double the supplemental buffer sizes if needed.
- Add a “v” prefix to instruction names.
- Change register names from xmm to ymm.
- Add destination registers to computational Intel AVX instructions.

- Divide the iteration count by two (or double stride length).

Example 15-2. Cartesian Coordinate Transformation with Assembly

```
//Use SSE Assembly
int main()
{
  int len = 3200;
  //Dynamic memory allocation with 16byte
  //alignment
  float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 16);
  float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 16);

//init data
for(int i=0; i<len; i++)
      pInVector[i] = 1;
  //Static memory allocation of 4 floats
  //with 16byte alignment
      float cos_theta = 0.8660254037;
      float sin_theta = 0.5;
  __declspec(align(16)) float cos_sin_theta_vec[4] =
{cos_theta, sin_theta, cos_theta, sin_theta};
__declspec(align(16)) float sin_cos_theta_vec[4] =
{sin_theta, cos_theta, sin_theta, cos_theta};

//processing 8 elements in an unrolled-twice loop
__asm
{
     mov rax, pInVector
     mov rbx, pOutVector
// Load into an xmm register of 16 bytes
     movups xmm3,
     xmmword ptr[cos_sin_theta_vec] movups xmm4,
     xmmword ptr[sin_cos_theta_vec] mov rdx, len
          shl rdx, 2       //size of input array in bytes
     xor rcx, rcx
```

```
// Use Intel AVX assembly
int main()
{
 int len = 3200;
 //Dynamic memory allocation with 32byte
 //alignment
 float* pInVector = (float*)
_mm_malloc(len*sizeof(float), 32);
 float* pOutVector = (float*)
_mm_malloc(len*sizeof(float), 32);

//init data
for(int i=0; i<len; i++)
   pInVector[i] = 1;

//Static memory allocation of 8 floats
 //with 32byte alignment
 float cos_theta = 0.8660254037;
 float sin_theta = 0.5;
 __declspec(align(32)) float cos_sin_theta_vec[8] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};

//processing 16 elements in an unrolled-twice loop
__asm
 {
   mov rax, pInVector
   mov rbx, pOutVector
// Load into an ymm register of 32 bytes
   vmovups ymm3,
         ymmword ptr[cos_sin_theta_vec]
   vmovups ymm4,
         ymmword ptr[sin_cos_theta_vec]
mov rdx, len
  shl rdx, 2//size of input array in bytes
   xor rcx, rcx
```

Example 15-2. Cartesian Coordinate Transformation with Assembly (Contd.)

```
loop1:
    movsldup  xmm0, [rax+rcx]
    movshdup  xmm1, [rax+rcx]
//example: mulps has 2 operands
    mulps xmm0, xmm3
    mulps xmm1, xmm4
    addsubps xmm0, xmm1
// 16 byte store from an xmm register
    movaps [rbx+rcx], xmm0
    movsldup  xmm0, [rax+rcx+16]
    movshdup  xmm1, [rax+rcx+16]
    mulps xmm0, xmm3
    mulps xmm1, xmm4
    addsubps xmm0, xmm1

// offset of 16 bytes from previous store
    movaps [rbx+rcx+16], xmm0
// Processed 32bytes in this loop
//(The code is unrolled twice)
add rcx, 32
    cmp rcx, rdx
    jl loop1
}


_mm_free(pInVector);
 _mm_free(pOutVector);
return 0;
}
```

```
loop1:
   vmovsldup  ymm0, [rax+rcx]
   vmovshdup  ymm1, [rax+rcx]
//example: vmulps has 3 operands
   vmulps ymm0, ymm0, ymm3
   vmulps ymm1, ymm1, ymm4
   vaddsubps ymm0, ymm0, ymm1
// 32 byte store from an ymm register
  vmovaps [rbx+rcx], ymm0
  vmovsldup  ymm0, [rax+rcx+32]
  vmovshdup  ymm1, [rax+rcx+32]
  vmulps ymm0, ymm0, ymm3
  vmulps ymm1, ymm1, ymm4
  vaddsubps ymm0, ymm0, ymm1

// offset of 32 bytes from previous store
   vmovaps [rbx+rcx+32], ymm0
// Processed 64bytes in this loop
//(The code is unrolled twice)
add rcx, 64
  cmp rcx, rdx
  jl loop1
  }



_mm_free(pInVector);
 _mm_free(pOutVector);
 return 0;
}
```

15.2 NON-DESTRUCTIVE SOURCE (NDS)

Most Intel AVX instructions have three operands. A typical instruction has two sources and one destination, with both source operands unmodified by the instruction. This section describes how using the NDS feature to save register copies, reduce the amount of instructions, reduce the amount of micro-ops, and improve performance. In this example, the Intel AVX code is more than 2x faster than the Intel SSE code.

The following example uses a vectorized calculation of the polynomial A^3+A^2+A. The polynomial calculation pseudo code is:

```
While (i<len)
        {
            B[i] := A[i]^3 + A[i]^2 + A[i]
            i++
        }
```

In Example 15-3, the left column shows the vectorized implementation using Intel SSE assembly. In this code, A is copied by an additional load from memory to a register, and A2 is copied using a register to register assignment. The code uses ten micro-ops to process four elements.

The middle column in this example uses 128-bit Intel AVX instructions and takes advantage of NDS. The additional load and register copies are eliminated. This code uses eight micro-ops to process four elements and is about 30% faster than the baseline above.

The right column in this example uses 256-bit AVX instructions. It uses eight micro-ops to process eight elements. Combining the NDS feature with the doubling of vector width, this speeds up the baseline by more than 2x.

Example 15-3. Direct Polynomial Calculation

SSE Code	128-bit AVX Code	256-bit AVX Code

float* pA = InputBuffer;
float* pB = OutputBuffer;
int len = miBufferWidth-4;
 	
__asm
{
mov rax, pA
mov rbx, pB
movsxd r8, len
	float* pA = InputBuffer1;
	float* pB = OutputBuffer1;
	int len = miBufferWidth-4;
	
	__asm
	{
	mov rax, pA
	mov rbx, pB
	movsxd r8, len
		float* pA = InputBuffer1;
		float* pB = OutputBuffer1;
		int len = miBufferWidth-8;
		
		__asm
		{
		mov rax, pA
		mov rbx, pB
		movsxd r8, len

loop1:
//Load A
movups xmm0, [rax+r8*4]
//Copy A
movups xmm1, [rax+r8*4]
//A^2
mulps xmm1, xmm1
//Copy A^2
movupsxmm2, xmm1
//A^3
mulps xmm2, xmm0
//A + A^2
addps xmm0, xmm1
//A + A^2 + A^3
addps xmm0, xmm2
//Store result
movups[rbx+r8*4], xmm0
sub r8, 4
jge loop1
}
	loop1:
	//Load A
	vmovups xmm0, [rax+r8*4]
	
	//A^2
	vmulps xmm1, xmm0, xmm0
	
	//A^3
	vmulps xmm2, xmm1, xmm0
	//A+A^2
	vaddps xmm0, xmm0, xmm1
	//A+A^2+A^3
	vaddps xmm0, xmm0, xmm2
	//Store result
	vmovups[rbx+r8*4], xmm0
	
	sub r8, 4
	jge loop1
	}
		loop1:
		//Load A
		vmovups ymm0, [rax+r8*4]
		
		//A^2
		vmulps ymm1, ymm0, ymm0
		
		//A^3
		vmulps ymm2, ymm1, ymm0
		//A+A^2
		vaddps ymm0, ymm0, ymm1
		//A+A^2+A^3
		vaddps ymm0, ymm0, ymm2
		//Store result
		vmovups [rbx+r8*4], ymm0
		sub r8, 8
		jge loop1
		}

15.3 MIXING AVX CODE WITH SSE CODE

The Intel AVX architecture allows programmers to port a large code base gradually, resulting in mixed Intel AVX and Intel SSE code. If your code includes both Intel AVX and Intel SSE, consider the following:

- Recompilation of Intel SSE code with the Intel compiler and the option “/QxAVX” in Windows or “-xAVX” in Linux. This transforms all Intel SSE instructions to 128-bit AVX instructions automatically. This refers to inline assembly and intrinsic code. “GCC -c -mAVX” will generate AVX code, including assembly files. GCC assembler also supports “-msse2avx” switch to generate AVX code from Intel SSE.

- Intel AVX and Intel SSE code can co-exist and execute in the same run. This can happen if your application includes third party libraries with Intel SSE code, a new DLL using Intel AVX code is deployed that calls other modules running Intel SSE code, or you cannot recompile all your application at once. In these cases, the Intel AVX code must use the VZEROUPPER instruction to avoid AVX/SSE transition penalty.

Intel AVX instructions always modify the upper bits of YMM registers and Intel SSE instructions do not modify the upper bits. From a hardware perspective, the upper bits of the YMM register collection can be considered to be in one of three states:

- Clean: All upper bits of YMM are zero. This is the state when the processor starts from RESET.
- Modified and Unsaved (In Table 15-2, this is abbreviated as M/U): The execution of one Intel AVX instruction (either 256-bit or 128-bit) modifies the upper bits of the destination YMM. This is also referred to as dirty upper YMM state. In this state, bits 255:128 and bits 127:0 of a given YMM are related to the most recent 256-bit or 128-bit AVX instruction that operated on that register.
- Preserved/Non_INIT Upper State (In Table 15-2, this is abbreviated as P/N): In this state, the upper YMM state is not zero. The upper 128 bits of a YMM and the lower 128 bits may be unrelated to the last Intel AVX instruction executed in the processor as a result of XRSTOR from a saved image with dirty upper YMM state.

If software inter-mixes Intel AVX and Intel SSE instructions without using VZEROUPPER properly, it can experience an Intel AVX/Intel SSE transition penalty. The situations of executing Intel SSE, Intel AVX, or managing the YMM state using XSAVE/XRSTOR/VZEROUPPER/VZEROALL is illustrated in Figure 15-1. The penalty associated with transitions into or out of the processor state “Modified and Unsaved” is implementation specific, depending on the microarchitecture.

Figure 15-1 depicts the situations that a transition penalty will occur for earlier microarchitectures that support Intel AVX. The transition penalty of A and B occurs with each instruction execution that would cause the transition. It is largely the cost of copying the entire YMM state to internal storage.

To minimize the occurrence of YMM state transitions related to the “Preserved/Non_INIT Upper State”, software that uses XSAVE/XRSTOR family of instructions to save/restore the YMM state should write a “Clean” upper YMM state to the XSAVE region in memory. Restoring a dirty YMM image from memory into the YMM registers can experience a penalty. This is illustrated in Figure 15-1.

The Skylake microarchitecture implements a different state machine than prior generations to manage the YMM state transition associated with mixing Intel SSE and Intel AVX instructions. It no longer saves the entire upper YMM state when executing an Intel SSE instruction when in “Modified and Unsaved” state, but saves the upper bits of individual register. As a result, mixing Intel SSE and Intel AVX instructions will experience a penalty associated with partial register dependency of the destination registers being used and additional blend operation on the upper bits of the destination registers. Figure 15-2 depicts the transition penalty applicable to the Skylake microarchitecture.

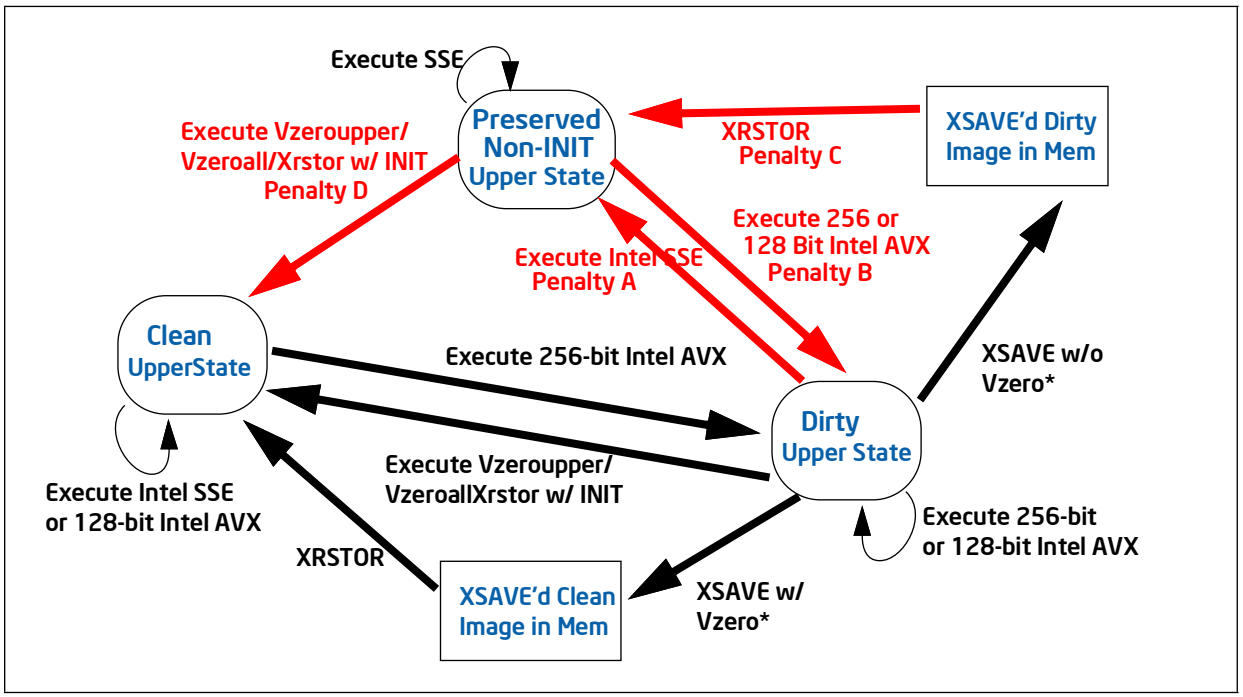


Figure 15-1. Intel® AVX—Intel® SSE Transitions in Earlier Microarchitectures

Table 15-2 lists the effect of mixing Intel AVX and Intel SSE code, with the bottom row indicating the types of penalty that might arise depending on the initial YMM state (the row marked 'Begin') and the ending state. Table 15-2 also includes the effect of transition penalty (Type C and D) associated with restoring a dirty YMM state image stored in memory.

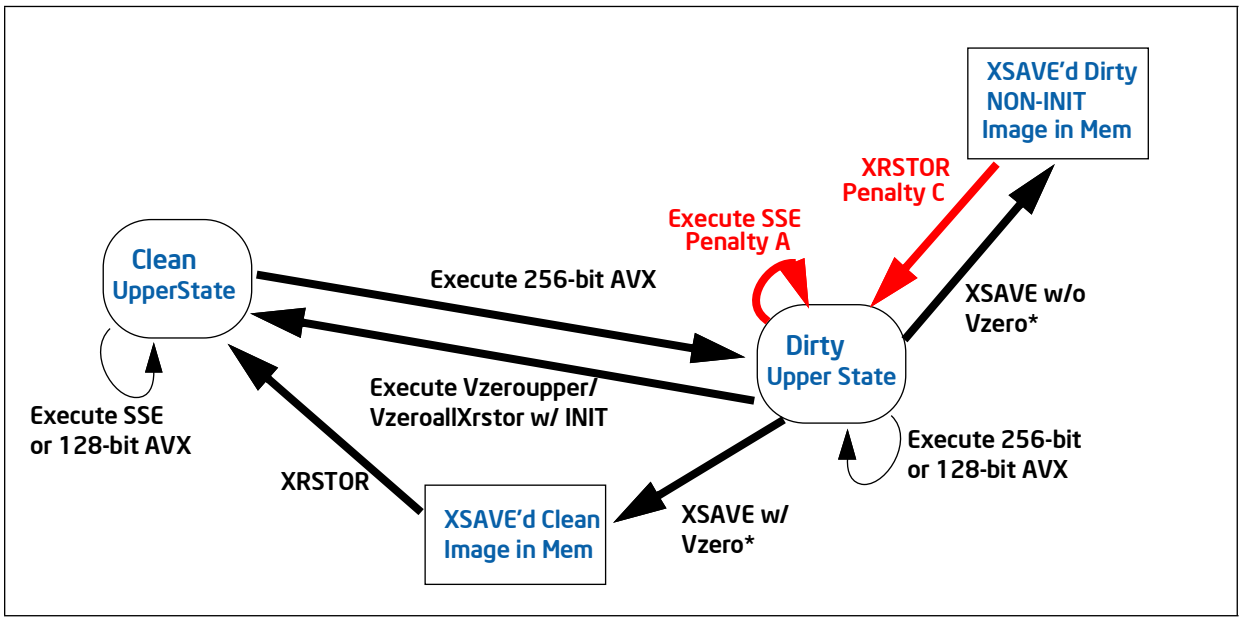


Figure 15-2. Intel® AVX- Intel® SSE Transitions in the Skylake Microarchitecture

Table 15-2. State Transitions of Mixing AVX and SSE Code

	Execute SSE			Execute AVX-128			Execute AVX-256			VZeroupper	XRSTOR	
Begin	Clean	M/U	P/N	Clean	M/U	P/S	Clean	M/U	P/N	P/N	Dirty Image	Clean Image
End	Clean	P/N	P/N	Clean	M/U	M/U	M/U	M/U	M/U	Clean	P/N	Clean
Penalty	No	A	No	No	No	B	No	No	B	D	C	No

The magnitude of each type of transition penalty can vary across different microarchitectures. In Skylake microarchitecture, some of the transition penalty is reduced. The transition diagram and associated penalty is depicted in Table 15-2. Table 15-3 gives approximate order of magnitude of the different transition penalty types across recent microarchitectures.

Table 15-3. Approximate Magnitude of Intel® AVX—Intel® SSE Transition Penalties in Different Microarchitectures

Type	Haswell Microarchitecture	Broadwell Microarchitecture	Skylake Microarchitecture	Ice Lake Client Microarchitecture
A	~XSAVE	~XSAVE	Partial Register Dependency + Blend	~XSAVE
B	~XSAVE	~XSAVE	NA	~XSAVE
C	~Fraction of XSAVE	~Fraction of XSAVE	~XSAVE	~Fraction of XSAVE
D	~XSAVE	~XSAVE	NA	~XSAVE

To enable fast transitions between 256-bit Intel AVX and Intel SSE code blocks, use the VZEROUPPER instruction before and after an AVX code block that would need to switch to execute SSE code. The VZEROUPPER instruction resets the upper 128 bits of all Intel AVX registers. This instruction has zero latency. In addition, the processor changes back to a Clean state, after which execution of SSE instructions or Intel AVX instructions has no transition penalty with prior microarchitectures. In Skylake microarchitecture, the SSE block can executed from a Clean state without the penalty of upper-bits dependency and blend operation.

128-bit Intel AVX instructions zero the upper 128-bits of the destination registers. Therefore, 128-bit and 256-bit Intel AVX instructions can be mixed with no penalty.

Assembly/Compiler Coding Rule 63. (H impact, H generality) *Whenever a 256-bit AVX code block and 128-bit SSE code block might execute in sequence, use the VZEROUPPER instruction to facilitate a transition to a "Clean" state for the next block to execute from.*

15.3.1 MIXING INTEL® AVX AND INTEL SSE IN FUNCTION CALLS

Intel AVX to Intel SSE transitions can occur unexpectedly when calling functions or returning from functions. For example, if a function that uses 256-bit Intel AVX, calls another function, the callee might be using SSE code. Similarly, after a 256-bit Intel AVX function returns, the caller might be executing Intel SSE code.

(H impact, H generality) *Add VZEROUPPER instruction after 256-bit AVX instructions are executed and before any function call that might execute SSE code. Add VZEROUPPER at the end of any function that uses 256-bit AVX instructions.*

Example 15-4. Function Calls and Intel® AVX/Intel® SSE transitions

```
__attribute__((noinline)) void SSE_function()
{
   __asm addps xmm1, xmm2
   __asm xorps xmm3, xmm4
}

__attribute__((noinline)) void AVX_function_no_zeroupper()
{
   __asm vaddps ymm1, ymm2, ymm3
   __asm vxorps ymm4, ymm5, ymm6
}
__attribute__((noinline)) void AVX_function_with_zeroupper()
{
   __asm vaddps ymm1, ymm2, ymm3
   __asm vxorps ymm4, ymm5, ymm6
 //add vzeroupper when returning from an AVX function
   __asm vzeroupper
}
```

```
// Code encounter transition penalty

__asm vaddps ymm1, ymm2, ymm3
..

//penalty
SSE_function();
AVX_function_no_zeroupper();
//penalty
__asm addps xmm1, xmm2
```

```
// Code mitigated transition penalty

__asm vaddps ymm1, ymm2, ymm3
//add vzeroupper before
//calling SSE function from AVX code
__asm vzeroupper   //no penalty
SSE_function();
AVX_function_with_zeroupper();
//no penalty
__asm addps xmm1, xmm2
```

Table 15-2 summarizes a heuristic of the performance impact of using or not using VZEROUPPER to bridge transitions of inter-function calls that changes between AVX code implementation and SSE code.

Table 15-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code

Inter-Function Call	Prior Microarchitectures	Skylake Microarchitecture
With VZEROUPPER	1X (baseline)	~1
No VZEROUPPER	< 0.1X	Fraction of baseline

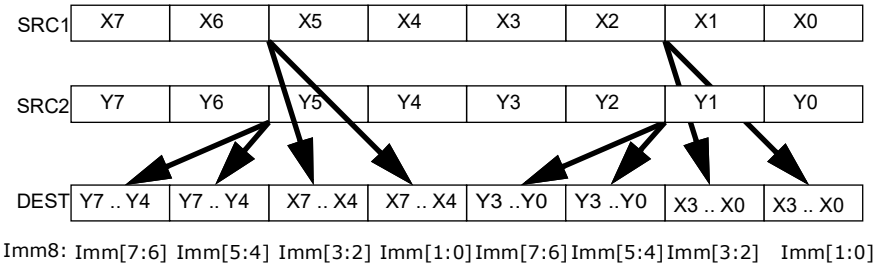
15.4 128-BIT LANE OPERATION AND AVX

256-bit operations in Intel AVX are generally performed in two halves of 128-bit lanes. Most of the 256-bit Intel AVX instructions are defined as in-lane: the destination elements in each lane are calculated using source elements only from the same lane. There are only a few cross-lane instructions, which are described below.

Most SSE computational instructions perform computation along vertical slots with each data elements. The 128-bit lanes does not affect porting 128-bit code into 256-bit AVX code. VADDPS is one example of this.

Many 128-bit SSE instruction moves data elements horizontally, e.g. SHUFPS uses an imm8 byte to control the horizontal movement of data elements.

Intel AVX promotes these horizontal 128-bit SIMD instruction in-lane into 256-bit operation by using the same control field within the low 128-bit land and the high 128-bit lane. For example, the 256-bit VSHUFPS instruction uses a control byte containing 4 control values to select the source location of each destination element in a 128-bit lane. This is shown below.



Control Values 00b: X0/Y0 (Low lane), X4/Y4 (high lane)
Control Values 01b: X1/Y1 (Low lane), X5/Y5 (high lane)
Control Values 10b: X2/Y2 (Low lane), X6/Y6 (high lane)
Control Values 11b: X3/Y3 (Low lane), X7/Y7 (high lane)

Figure 15-3. Source Location of Each Destination Element in a 128-bit Lane

15.4.1 PROGRAMMING WITH THE LANE CONCEPT

Using the lane concept, algorithms implemented with SSE instruction set can be easily converted to use 256-bit Intel AVX. An SSE algorithm that executes iterations 0 to n can be converted so the calculation of iteration i is done in the low lane and the calculation of iteration i+k is done in the high lane. For consecutive iterations k equals one.

Some vectorized algorithms implemented with SSE instructions cannot use a simple conversion described above. For example, shuffles that move elements within 16 bytes cannot be naturally converted to shuffles with 32 byte since 32 byte shuffles can't cross lanes.

You can use the following instructions as building blocks for working with lanes:

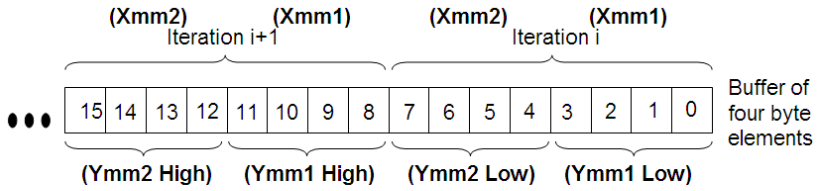
- VINSERTF128 - insert packed floating-point values.
- VEXTRACTF128 - extract packed floating-point values.
- VPERM2F128 - permute floating-point values.
- VBROADCAST - load with broadcast.

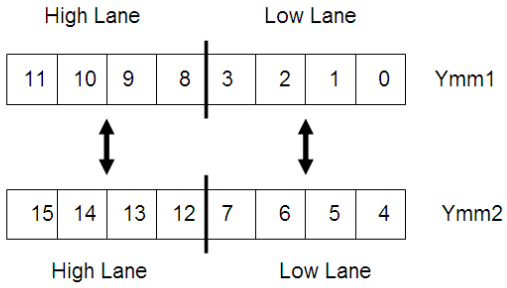
The sections below describe two techniques: the strided loads and the cross register overlap. These methods implement the in lane data arrangement described above and are useful in many algorithms that initially seem to require cross lane calculations.

15.4.2 STRIDED LOAD TECHNIQUE

The strided load technique is a programming method that uses Intel AVX instructions and is useful for algorithms that involve unsupported cross-lane shuffles.

The method describes how to arrange data to avoid cross-lane shuffles. The main idea is to use 128-bit loads in a way that mimics the corresponding Intel SSE algorithm, and enables the 256 Intel AVX instructions to execute iterations i of the loop in the low lanes and the iteration and i+k in the high lanes. In the following example, k equals one.





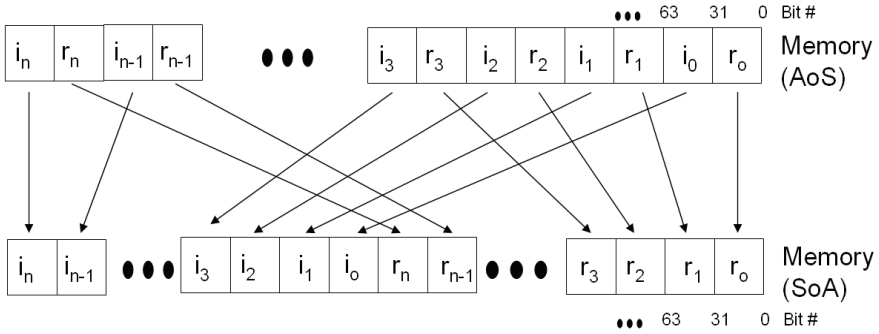
The values in the low lanes of Ymm1 and Ymm2 in the figure above correspond to iteration i in the SSE implementation. Similarly, the values in the high lanes of Ymm1 and Ymm2 correspond to iteration i+1.

Example 15-5 demonstrates the strided load method in a conversion of an Array of Structures (AoS) to a Structure of

Example 15-5. AoS to SoA Conversion of Complex Numbers in C Code

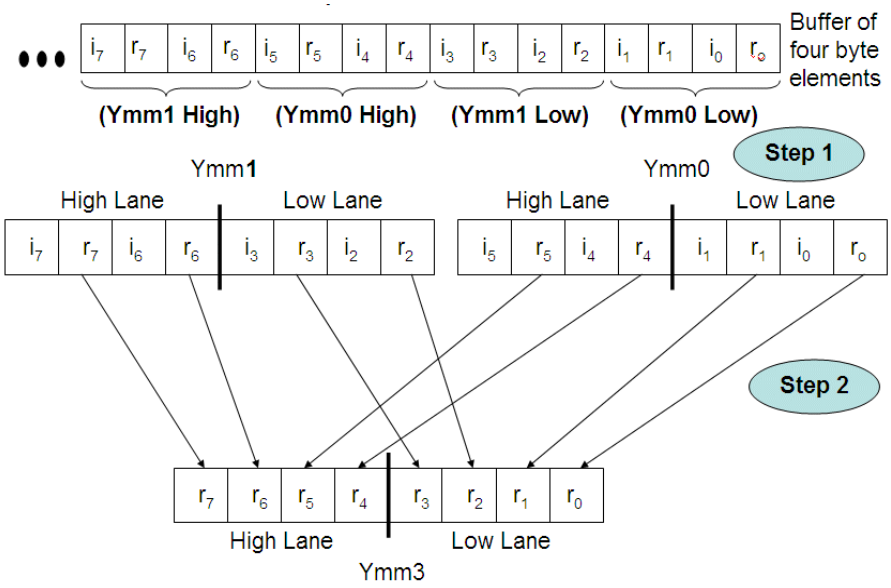
```
for (i = 0; i < N; i++)
{
  Real[i] = Complex[i].Real;
  Imaginary[i] = Complex[i].Imaginary;
}
```

Arrays (SoA). In this example, the input buffer contains complex numbers in an AoS format. Each complex number is made of a real and an imaginary float values. The output buffer is arranged as SoA. All the real components of the complex numbers are located at the first half of the output buffer and all the imaginary components are located at the second half of the buffer. The following pseudo code and figure illustrate the conversion:



A simple extension of the Intel SSE algorithm from 16-byte to 32-byte operations would require cross-lane data transition, as shown in the following figure. However, this is not possible with Intel AVX architecture and a different technique is required.

The challenge of cross-lane shuffle can be overcome with Intel AVX for AoS to SoA conversion. Using VINSERTF128 to load 16 bytes to the appropriate lane in the YMM registers obviates the need for cross-lane shuffle. Once the data is organized properly in the YMM registers for step 1, 32-byte VSHUFPS can be used to move the data in lanes, as shown in step 2.



The following code compares the Intel SSE implementation of AoS to SoA with the 256-bit Intel AVX implementation and demonstrates the performance gained.

Example 15-6. Aos to SoA Conversion of Complex Numbers Using Intel® AVX

Intel® SSE Code	Intel® AVX Code
xor rbx, rbx	xor rbx, rbx
xor rdx, rdx	xor rdx, rdx
mov rcx, len	mov rcx, len
mov rdi, inPtr	mov rdi, inPtr
mov rsi, outPtr1	mov rsi, outPtr1
mov rax, outPtr2	mov rax, outPtr2
loop1:	loop1:
movups xmm0, [rdi+rbx]	vmovups xmm0, [rdi+rbx]
//i1 r1 i0 r0	//i1 r1 i0 r0
movups xmm1, [rdi+rbx+16]	vmovups xmm1, [rdi+rbx+16]
// i3 r3 i2 r2	// i3 r3 i2 r2
movups xmm2, xmm0	vinsertf128 ymm0, ymm0, [rdi+rbx+32] , 1
	//i5 r5 i4 r4; i1 r1 i0 r0
shufps xmm0, xmm1, 0xdd	vinsertf128 ymm1, ymm1, [rdi+rbx+48] , 1
//i3 i2 i1 i0	//i7 r7 i6 r6; i3 r3 i2 r2
shufps xmm2, xmm1, 0x88	vshufps ymm2, ymm0, ymm1, 0xdd
//r3 r2 r1 r0	//i7 i6 i5 i4; i3 i2 i1 i0
	vshufps ymm3, ymm0, ymm1, 0x88
	//r7 r6 r5 r4; r3 r2 r1 r0
movups [rax+rdx], xmm0	vmovups [rax+rdx], ymm2

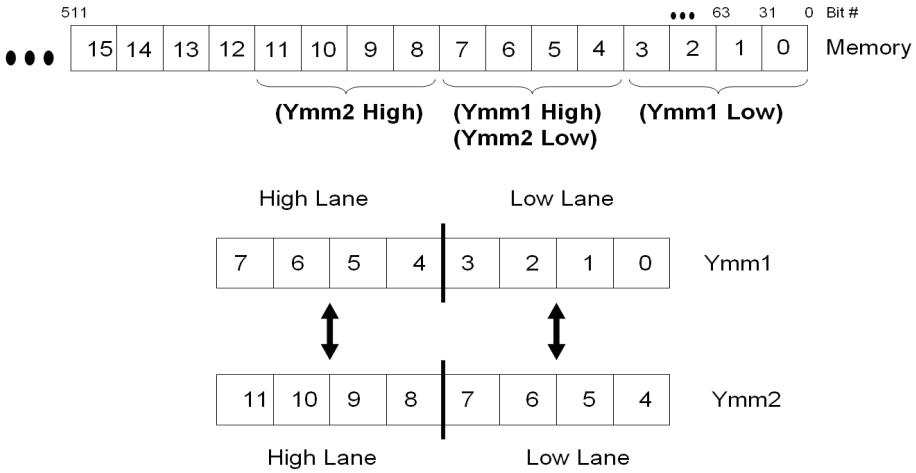
Example 15-6. Aos to SoA Conversion of Complex Numbers Using Intel® AVX (Contd.)

Intel® SSE Code	Intel® AVX Code
movups [rsi+rdx], xmm2	vmovups [rsi+rdx], ymm3
add rdx, 16	add rdx, 32
add rbx, 32	add rbx, 64
cmp rcx, rbx	cmp rcx, rbx
jnz loop1	jnz loop1

15.4.3 THE REGISTER OVERLAP TECHNIQUE

The register overlap technique is useful for algorithms that use shuffling. Similar to the strided load technique, the register overlap technique arranges data to avoid cross-lane shuffles.

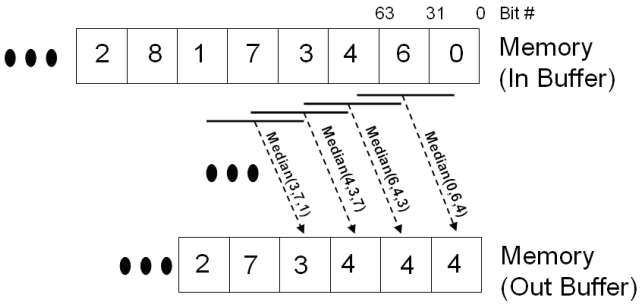
This technique is useful for algorithm that process continues data, which is partially shared by sequential iterations. The following figure illustrates the desired data layout. This is enabled by using overlapping 256-bit loads, or by using the VPERM2F128 instruction.



The Median3 code sample below demonstrates the register overlap technique. The median3 technique calculates the median of every three consecutive elements in a vector.

Y[i] = Median(X[i], X[i+1], X[i+2])

Where Y is the output vector and X is the input vector. The following figure illustrates the calculation done by the median algorithm.



Following are three implementations of the Median3 algorithm:

- **Alternative 1:** Intel SSE implementation.
- **Alternatives 2 and 3**: Register overlap technique in two ways.
  - — Alternative 2 loads the data from the input buffer into the YMM registers using overlapping 256-bit load operations.
  - — Alternative 3 loads the data from the input buffer into the YMM registers using a 256-bit load operation and VPERM2F128.
  - — Alternatives 2 and 3 gain performance by using wider vectors.

Example 15-7. Register Overlap Method for Median of 3 Numbers

1: SSE Code	2: 256-bit AVX w/Overlapping Loads	3: 256-bit AVX with VPERM2F128

xor ebx, ebx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
movaps xmm0, [rdi]
 	
loop_start:
movaps xmm4, [rdi+16]
movaps xmm2, [rdi]
movaps xmm1, [rdi]
movaps xmm3, [rdi]
 	
add rdi, 16
add rbx, 4
shufps xmm2, xmm4, 0x4e
shufps xmm1, xmm2, 0x99
minps xmm3, xmm1
maxps xmm0, xmm1
minps xmm0, xmm2
maxps xmm0, xmm3
movaps [rsi], xmm0
movaps xmm0, xmm4
add rsi, 16
cmp rbx, rcx
jl loop_start
	xor ebx, ebx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr
	vmovaps ymm0, [rdi]
	
	loop_start:
	vshufps ymm2, ymm0,
	[rdi+16], 0x4E
	vshufps ymm1, ymm0,
	ymm2, 0x99
	
	add rbx, 8
	add rdi, 32
	vminps ymm4, ymm0, ymm1
	vmaxps ymm0, ymm0, ymm1
	vminps ymm3, ymm0, ymm2
	vmaxps ymm5, ymm3, ymm4
	vmovaps [rsi], ymm5
	add rsi, 32
	vmovaps ymm0, [rdi]
	cmp rbx, rcx
	jl loop_start
		xor ebx, ebx
		mov rcx, len
		mov rdi, inPtr
		mov rsi, outPtr
		vmovaps ymm0, [rdi]
		
		loop_start:
		add rdi, 32
		vmovaps ymm6, [rdi]
		vperm2f128 ymm1, ymm0, ymm6, 0x21
		vshufps ymm3, ymm0, ymm1, 0x4E
		
		vshufps ymm2, ymm0, ymm3, 0x99
		add rbx, 8
		vminps ymm5, ymm0, ymm2
		vmaxps ymm0, ymm0, ymm2
		vminps ymm4, ymm0, ymm3
		vmaxps ymm7, ymm4, ymm5
		vmovaps ymm0, ymm6
		vmovaps [rsi], ymm7
		add rsi, 32
		cmp rbx, rcx
		jl loop_start

15.5 DATA GATHER AND SCATTER

This section describes techniques for implementing data gather and scatter operations using Intel AVX instructions.

15.5.1 DATA GATHER

The gather operation reads elements from an input buffer based on indexes specified in an index buffer. The gathered elements are written in an output buffer. The following figure illustrates an example for a gather operation.

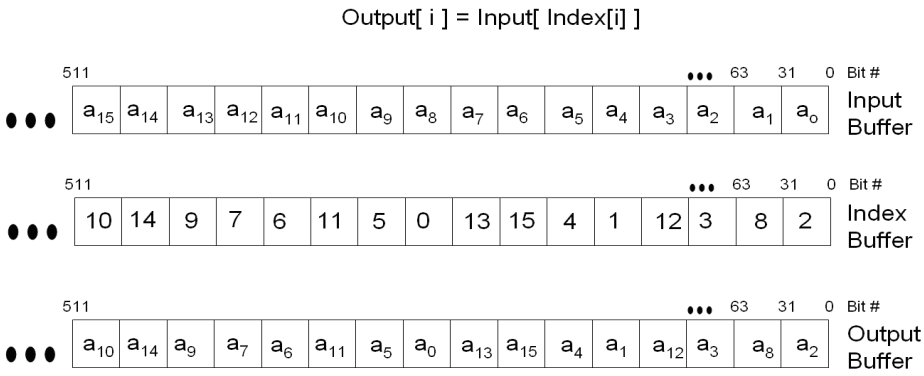
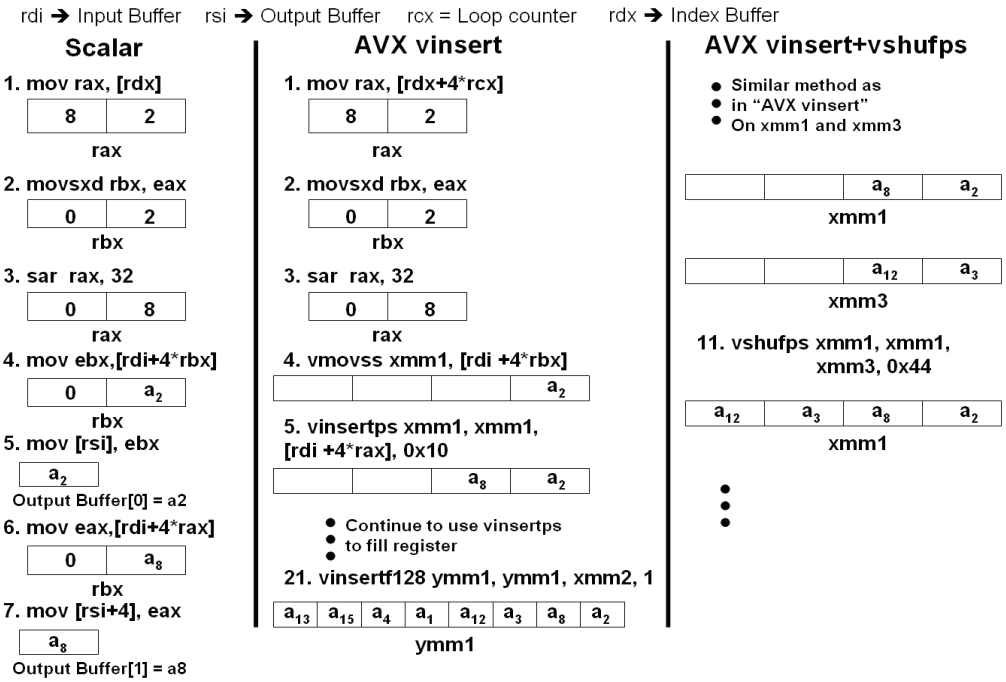


Figure 15-4. Gather Operation

Following are 3 implementations for the gather operation from an array of 4-byte elements.

- Alternative one is a scalar implementation using general purpose registers.
- Alternatives two and three use Intel AVX instructions.

The figure below shows code snippets from Example 15-8 assuming that it runs the first iteration on data from the previous figure.



Performance of the Intel AVX examples is similar to the performance of a corresponding Intel SSE implementation. The table below shows the three gather implementations.

Example 15-8. Data Gather - Intel® AVX versus Scalar Code

1: Scalar Code	2: Intel® AVX w/ VINSERTPS	3: VINSERTPS+VSHUFPS

mov rdi, InBuf
mov rsi, OutBuf
mov rdx, Index
xor rcx, rcx
 	
loop1:
mov rax, [rdx]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi + 4*rbx]
mov [rsi], ebx
mov eax, [rdi + 4*rax]
mov [rsi + 4], eax
mov rax, [rdx + 8]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi + 4*rbx]
mov [rsi + 8], ebx
mov eax, [rdi + 4*rax]
mov [rsi + 12], eax
	mov rdi, InBuf
	mov rsi, OutBuf
	mov rdx, Index
	xor rcx, rcx
	
	loop1:
	mov rax, [rdx + 4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vmovss xmm1, [rdi + 4*rbx]
	vinsertps xmm1, xmm1, [rdi + 4*rax], 0x10
	mov rax, [rdx + 8 + 4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vinsertps xmm1, xmm1, [rdi + 4*rbx], 0x20
	vinsertps xmm1, xmm1, [rdi + 4*rax], 0x30
		mov rdi, InBuf
		mov rsi, OutBuf
		mov rdx, Index
		xor rcx, rcx
		
		loop1:
		mov rax, [rdx + 4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm1, [rdi + 4*rbx]
		vinsertps xmm1, xmm1, [rdi + 4*rax], 0x10
		mov rax, [rdx + 8 + 4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm3, [rdi + 4*rbx]
		vinsertps xmm3, xmm3, [rdi + 4*rax], 0x10
		vshufps xmm1, xmm1, xmm3, 0x44

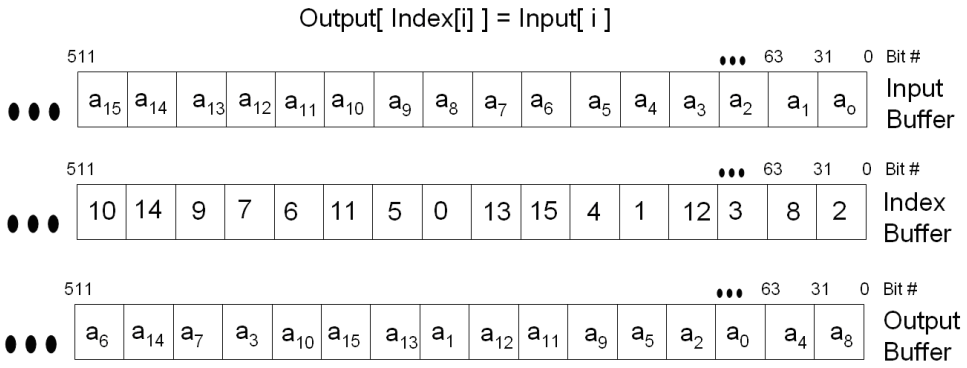
Example 15-8. Data Gather - Intel® AVX versus Scalar Code (Contd.)

1: Scalar Code	2: Intel® AVX w/ VINSERTPS	3: VINSERTPS+VSHUFPS

mov rax, [rdx + 16]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi + 4*rbx]
mov [rsi + 16], ebx
mov eax, [rdi + 4*rax]
mov [rsi + 20], eax
mov rax, [rdx + 24]
movsxd rbx, eax
sar rax, 32
mov ebx, [rdi + 4*rbx]
mov [rsi + 24], ebx
mov eax, [rdi + 4*rax]
mov [rsi + 28], eax
 	
add rsi, 32
add rdx, 32
add rcx, 8
cmp rcx, len
jl loop1
	mov rax, [rdx + 16 + 4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vmovss xmm2, [rdi + 4*rbx]
	vinsertps xmm2, xmm2, [rdi + 4*rax], 0x10
	
	mov rax, [rdx + 24 + 4*rcx]
	movsxd rbx, eax
	sar rax, 32
	vinsertps xmm2, xmm2, [rdi + 4*rbx], 0x20
	vinsertps xmm2, xmm2, [rdi + 4*rax], 0x30
	vinsertf128 ymm1, ymm1, xmm2, 1
	vmovaps [rsi + 4*rcx], ymm1
	
	add rcx, 8
	cmp rcx, len
	jl loop1
		mov rax, [rdx + 16 + 4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm2, [rdi + 4*rbx]
		vinsertps xmm2, xmm2, [rdi + 4* rax], 0x10
		
		mov rax, [rdx + 24 + 4*rcx]
		movsxd rbx, eax
		sar rax, 32
		vmovss xmm4, [rdi + 4*rbx]
		vinsertps xmm4, xmm4, [rdi + 4*rax], 0x10
		vshufps xmm2, xmm2, xmm4, 0x44
		vinsertf128 ymm1, ymm1, xmm2, 1
		vmovaps [rsi + 4*rcx], ymm1
		add rcx, 8
		cmp rcx, len
		jl loop1

15.5.2 DATA SCATTER

The scatter operation reads elements from an input buffer sequentially. It then writes them to an output buffer based on indexes specified in an index buffer. The following figure illustrates an example for a scatter operation.



The following example includes a scalar implementation and an Intel AVX implementation of a scatter sequence. The Intel AVX examples consist mainly of 128-bit Intel AVX instructions. Performance of the Intel AVX examples is similar to the performance of corresponding Intel SSE implementation.

Example 15-9. Scatter Operation Using Intel® AVX

Scalar Code	AVX Code
movrdi, InBuf movrsi, OutBuf movrdx, Index xor rcx, rcx loop1: movsxd rax, [rdx] mov ebx, [rdi] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 4] mov ebx, [rdi + 4] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 8] mov ebx, [rdi + 8] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 12] mov ebx, [rdi + 12] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 16] mov ebx, [rdi + 16] mov [rsi + 4*rax], ebx movsxd rax, [rdx + 20]	movrdi, InBuf movrsi, OutBuf movrdx, Index xor rcx, rcx loop1: vmovaps ymm0, [rdi + 4*rcx] movsxd rax, [rdx + 4*rcx] movsxd rbx, [rdx + 4*rcx + 4] vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 8] vpalignr xmm1, xmm0, xmm0, 4 vmovss [rsi + 4*rbx], xmm1 movsxd rbx, [rdx + 4*rcx + 12] vpalignr xmm2, xmm0, xmm0, 8 vmovss [rsi + 4*rax], xmm2 movsxd rax, [rdx + 4*rcx + 16] vpalignr xmm3, xmm0, xmm0, 12 vmovss [rsi + 4*rbx], xmm3 movsxd rbx, [rdx + 4*rcx + 20] vextractf128 xmm0, ymm0, 1 vmovss [rsi + 4*rax], xmm0 movsxd rax, [rdx + 4*rcx + 24]

mov ebx, [rdi + 20]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 24]
mov ebx, [rdi + 24]
mov [rsi + 4*rax], ebx
movsxd rax, [rdx + 28]
mov ebx, [rdi + 28]
mov [rsi + 4*rax], ebx
add rdi, 32
add rdx, 32
add rcx, 8
cmprcx, len
jl loop1
	vpalignr xmm1, xmm0, xmm0, 4
	vmovss [rsi + 4*rbx], xmm1
	movsxd rbx, [rdx + 4*rcx + 28]vpalignr xmm2, xmm0, xmm0, 8
	vmovss [rsi + 4*rax], xmm2
	vpalignr xmm3, xmm0, xmm0, 12
	vmovss [rsi + 4*rbx], xmm3
	add rcx, 8
	cmprcx, len
	jl loop1

15.6 DATA ALIGNMENT FOR INTEL® AVX

This section explains the benefit of aligning data that is used by Intel AVX instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

The C code below is a C implementation of SAXPY.

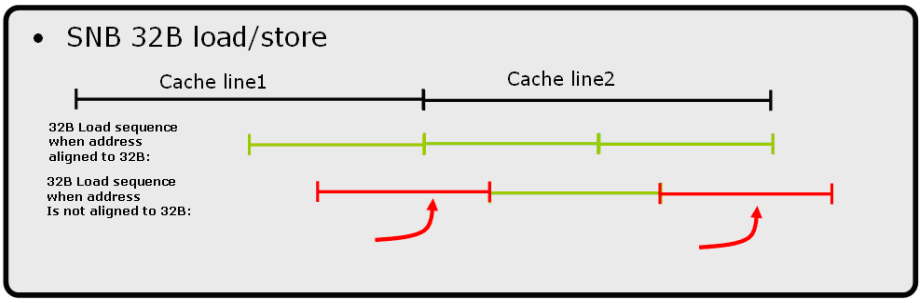
for (int i = 0; i < n; i++)

{ c[i] = alpha * a[i] + b[i]; }

15.6.1 ALIGN DATA TO 32 BYTES

Aligning data to vector length is recommended. When using 16-byte SIMD instructions, loaded data should be aligned to 16 bytes. Similarly, for best results when using Intel AVX instructions with 32-byte registers align the data to 32-bytes.

When using Intel AVX with unaligned 32-byte vectors, every second load is a cache-line split, since the cache-line is 64 bytes. This doubles the cache line split rate compared to Intel SSE code that uses 16-byte vectors. Even though split line access penalties have been reduced significantly since Nehalem microarchitecture, a high cache-line split rate in memory-intensive code may cause performance degradation.



Example 15-10. SAXPY using Intel® AVX

```
  mov  rax, src1
  mov  rbx, src2
  mov  rcx, dst
  mov  rdx, len
  xor    rdi, rdi
  vbroadcastss ymm0, alpha
start_loop:
  vmovupsymm1, [rax + rdi]
  vmulps ymm1, ymm1, ymm0
  vmovupsymm2, [rbx + rdi]
  vaddps  ymm1, ymm1, ymm2
  vmovups[rcx + rdi], ymm1
  vmovupsymm1, [ rax + rdi + 32]
  vmulps ymm1, ymm1, ymm0
  vmovups ymm2, [rbx + rdi + 32]
```

Example 15-10. SAXPY using Intel® AVX (Contd.)

```
vaddps ymm1, ymm1, ymm2
vmovups [rcx + rdi + 32], ymm1

add     rdi, 64
cmp     rdi, rdx
jl      start_loop
```

SAXPY is a memory intensive kernel that emphasizes the importance of data alignment. Optimal performance requires both data source address to be 32-byte aligned and destination address also 32-byte aligned[1].

- If only one of the three address is not aligned to 32-byte boundary, the performance may be halved.
- If all three addresses are mis-aligned relative to 32 byte, the performance degrades further.
- In some cases, unaligned accesses may result in lower performance for Intel AVX code compared to Intel SSE code.

Other Intel AVX kernels typically have more computation which can reduce the effect of the data alignment penalty.

Assembly/Compiler Coding Rule 64. (H impact, M generality) *Align data to 32-byte boundary when possible. Prefer store alignment over load alignment.*

- You can use dynamic data alignment using the **_mm_malloc** intrinsic instruction with the Intel® Compiler, or **_aligned_malloc** of the Microsoft® Compiler.
  For example:

```
//dynamically allocating 32byte aligned buffer with 2048 float elements.

InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 32);
```

- You can use static data alignment using __declspec(align(32)).
  For example:

```
//Statically allocating 32byte aligned buffer with 2048 float elements.

__declspec(align(32)) float InputBuffer[2048];
```

15.6.2 CONSIDER 16-BYTE MEMORY ACCESS WHEN MEMORY IS UNALIGNED

For best results use Intel AVX 32-byte loads and align data to 32-bytes. However, there are cases where you cannot align the data, or data alignment is unknown. This can happen when you are writing a library function and the input data alignment is unknown. In these cases, using 16-byte memory accesses may be the best alternative. The following method uses 16-byte loads while still benefiting from the 32-byte YMM registers.

NOTE

Beginning with Skylake microarchitecture, this optimization is not necessary. The only case where 16-byte loads may be more efficient is when the data is 16-byte aligned but not 32-byte aligned. In this case 16-byte loads might be preferable as no cache line split memory accesses are issued.

1. See Example 10 in GitHub.

Consider replacing unaligned 32-byte memory accesses using a combination of VMOVUPS, VINSERTF128, and VEXTRACTF128 instructions.

Example 15-11. Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation

```
Convert 32-byte loads as follows:
    vmovups      ymm0, mem ->   vmovups xmm0, mem
                                vinsertf128   ymm0, ymm0, mem+16, 1
Convert 32-byte stores as follows:
    vmovups mem, ymm0      ->   vmovups mem, xmm0
                                vextractf128 mem+16, ymm0, 1
```

The following intrinsics are available to handle unaligned 32-byte memory operating using 16-byte memory accesses:

```
    _mm256_loadu2_m128 ( float const * addr_hi, float const * addr_lo);
    _mm256_loadu2_m128d ( double const * addr_hi, double const * addr_lo);
    _mm256_loadu2_m128 i( __m128i const * addr_hi, __m128i const * addr_lo);
    _mm256_storeu2_m128 ( float * addr_hi, float * addr_lo, __m256 a);
    _mm256_storeu2_m128d ( double * addr_hi, double * addr_lo, __m256d a);
    _mm256_storeu2_m128 i( __m128i * addr_hi, __m128i * addr_lo, __m256i a);
```

Example 15-12[1] shows two implementations for SAXPY with unaligned addresses. Alternative one use 32-byte loads and alternative two uses 16-byte loads. These code samples are executed with two source buffers, src1, src2, at 4 byte offset from 32-byte alignment, and a destination buffer, DST, that is 32-byte aligned. Using two 16-byte memory operations in lieu of 32-byte memory access performs faster.[2]

Example 15-12. SAXPY Implementations for Unaligned Data Addresses

AVX with 32-byte memory operation	AVX using two 16-byte memory operations

mov rax, src1
mov rbx, src2
mov rcx, dst
mov rdx, len
xor rdi, rdi
vbroadcastss ymm0, alpha
start_loop:
vmovups ymm1, [rax + rdi]
vmulps ymm1, ymm1, ymm0
vmovups ymm2, [rbx + rdi]
vaddps ymm1, ymm1, ymm2
vmovups [rcx + rdi], ymm1
 	
vmovups ymm1, [rax+rdi+32]
vmulps ymm1, ymm1, ymm0
	mov rax, src1
	mov rbx, src2
	mov rcx, dst
	mov rdx, len
	xor rdi, rdi
	vbroadcastss ymm0, alpha
	start_loop:
	vmovups xmm2, [rax+rdi]
	vinsertf128 ymm2, ymm2, [rax+rdi+16], 1
	vmulps ymm1, ymm0, ymm2
	vmovups xmm2, [rbx + rdi]
	vinsertf128 ymm2, ymm2, [rbx+rdi+16], 1
	vaddps ymm1, ymm1, ymm2
	vmovups [rcx+rdi], ymm1
	vmovups xmm2, [rax+rdi+32]
	vinsertf128 ymm2, ymm2, [rax+rdi+48], 1
	vmulps ymm1, ymm0, ymm2

1. See Example 12 in GitHub.
2. Beginning with Haswell microarchitecture and onward, it is better to read the entire register: 32-byte register or 64-byte register (with the availability of Intel® AVX-512).

Example 15-12. SAXPY Implementations for Unaligned Data Addresses (Contd.)

AVX with 32-byte memory operation	AVX using two 16-byte memory operations

vmovups ymm2, [rbx+rdi+32]
vaddps ymm1, ymm1, ymm2
vmovups [rcx+rdi+32], ymm1
 	
add rdi, 64
cmp rdi, rdx
jl start_loop
	vmovups xmm2, [rbx+rdi+32]
	vinsertf128 ymm2, ymm2, [rbx+rdi+48], 1
	vaddps ymm1, ymm1, ymm2
	vmovups [rcx+rdi+32], ymm1
	add rdi, 64
	cmp rdi, rdx
	jl start_loop

Assembly/Compiler Coding Rule 65. (M impact, H generality) *Align data to 32-byte boundary when possible. If it is not possible to align both loads and stores, then prefer store alignment over load alignment.*

15.6.3 PREFER ALIGNED STORES OVER ALIGNED LOADS

There are cases where it is possible to align only a subset of the processed data buffers. In these cases, aligning data buffers used for store operations usually yields better performance than aligning data buffers used for load operations.

Unaligned stores are likely to cause greater performance degradation than unaligned loads, since there is a very high penalty on stores to a split cache-line that crosses pages. This penalty is estimated at 150 cycles. Stores that cross a page boundary are executed at retirement. In Example 15-12, unaligned store address can affect SAXPY performance for 3 unaligned addresses to about one quarter of the aligned case.

15.7 L1D CACHE LINE REPLACEMENTS

NOTE

Beginning with Haswell microarchitecture, cache line replacement is no longer a concern .

When a load misses the L1D Cache, a cache line with the requested data is brought from a higher memory hierarchy level. In memory intensive code where the L1D Cache is always active, replacing a cache line in the L1D Cache may delay other loads. In Sandy Bridge and Ivy Bridge microarchitectures, the penalty for 32-Byte loads may be higher than the penalty for 16-Byte loads. Therefore, memory intensive Intel AVX code with 32-Byte loads and with data set larger than the L1D Cache may be slower than similar code with 16-Byte loads.

When Example 15-12 is run with a data set that resides in the L2 Cache, the 16-byte memory access implementation is slightly faster than the 32-byte memory operation.

Be aware that the relative merit of 16-byte memory accesses versus 32-byte memory access is implementation specific across generations of microarchitectures.

In Haswell microarchitecture, the L1D Cache can support two 32-byte fetch each cycle.

15.8 4K ALIASING

4-KByte memory aliasing occurs when the code stores to one memory location and shortly after that it loads from a different memory location with a 4-KByte offset between them. For example, a load to linear address 0x400020 follows a store to linear address 0x401020.

The load and store have the same value for bits 5 - 11 of their addresses and the accessed byte offsets should have partial or complete overlap.

4K aliasing may have a five-cycle penalty on the load latency. This penalty may be significant when 4K aliasing happens repeatedly and the loads are on the critical path. If the load spans two cache lines it might be delayed until the conflicting store is committed to the cache. Therefore 4K aliasing that happens on repeated unaligned Intel AVX loads incurs a higher performance penalty.

To detect 4K aliasing, use the LD_BLOCKS_PARTIAL.ADDRESS_ALIAS event that counts the number of times Intel AVX loads were blocked due to 4K aliasing.

To resolve 4K aliasing, try the following methods in the following order:

- Align data to 32 Bytes.
- Change offsets between input and output buffers if possible.
- Sandy Bridge and Ivy Bridge microarchitectures may benefit from using 16-Byte memory accesses on memory which is not 32-Byte aligned.

15.9 CONDITIONAL SIMD PACKED LOADS AND STORES

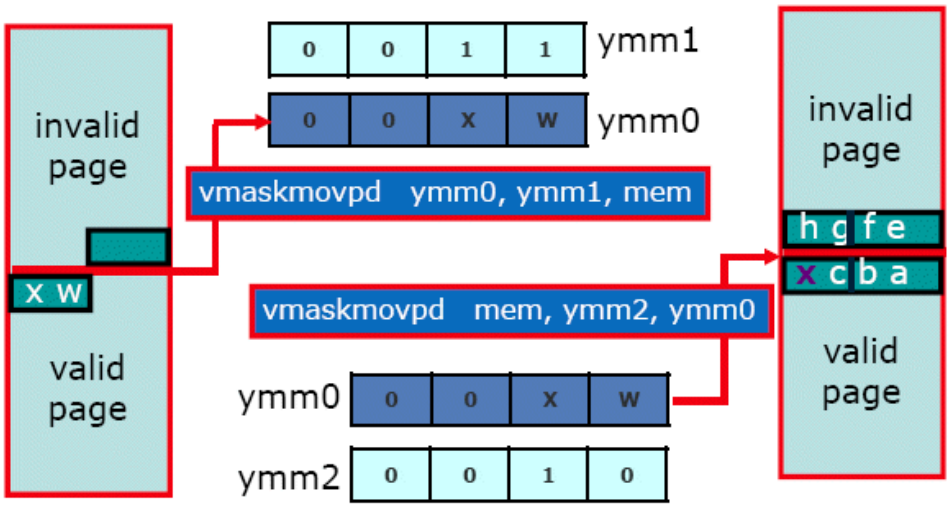
The VMASKMOV instruction conditionally moves packed data elements to/from memory, depending on the mask bits associated with each data element. The mask bit for each data element is the most significant bit of the corresponding element in the mask register.

When performing a mask load, the returned value is 0 for elements which have a corresponding mask value of 0. The mask store instruction writes to memory only the elements with a corresponding mask value of 1, while preserving memory values for elements with a corresponding mask value of 0. Faults can occur only for memory accesses that are required by the mask. Faults do not occur due to referencing any memory location if the corresponding mask bit value for that memory location is zero. For example, no faults are detected if the mask bits are all zero.

The following figure shows an example for a mask load and a mask store which does not cause a fault. In this example, the mask register for the load operation is ymm1 and the mask register for the store operation is ymm2.

When using masked load or store consider the following:

- On processors based on microarchitectures prior to Skylake, the address of a VMASKMOV store is considered as resolved only after the mask is known. Loads that follow a masked store may be blocked, depending on the memory disambiguation prediction, until the mask value is known.
- If the mask is not all 1 or all 0, loads that depend on the masked store have to wait until the store data is written to the cache. If the mask is all 1 the data can be forwarded from the masked store to the dependent loads. If the mask is all 0 the loads do not depend on the masked store.



- Masked loads including an illegal address range do not result in an exception if the range is under a zero mask value. However, the processor may take a multi-hundred-cycle “assist” to determine that no part of the illegal range have a one mask value. This assist may occur even when the mask is “zero” and it seems obvious to the programmer that the load should not be executed.

When using VMASKMOV, consider the following:

- Use VMASKMOV only in cases where VMOVUPS cannot be used.
- Use VMASKMOV on 32Byte aligned addresses if possible.
- If possible use valid address range for masked loads, even if the illegal part is masked with zeros.
- Determine the mask as early as possible.
- Avoid store-forwarding issues by performing loads prior to a VMASKMOV store if possible.
- Be aware of mask values that would cause the VMASKMOV instruction to require assist (if an assist is required, the latency of VMASKMOV to load data will increase dramatically):
  - Load data using VMASKMOV with a mask value selecting 0 elements from an illegal address will require an assist.
  - Load data using VMASKMOV with a mask value selecting 0 elements from a legal address expressed in some addressing form (e.g. [base+index], disp[base+index])will require an assist.

With processors based on the Skylake microarchitecture, the performance characteristics of VMASKMOV instructions have the following notable items:

- Loads that follow a masked store is not longer blocked until the mask value is known.
- Store data using VMASKMOV with a mask value permitting 0 elements to be written to an illegal address will require an assist.

15.9.1 CONDITIONAL LOOPS

VMASKMOV enables vectorization of loops that contain conditional code. There are two main benefits in using VMASKMOV over the scalar implementation in these cases:

- VMASKMOV code is vectorized.
- Branch mispredictions are eliminated.

Below is a conditional loop C code:

Example 15-13. Loop with Conditional Expression

```
for(int i = 0; i < miBufferWidth; i++)
{       if(A[i]>0)
        {               B[i] = (E[i]*C[i]);
        }
        else
        {               B[i] = (E[i]*D[i]);
        }
}
```

Example 15-14. Handling Loop Conditional with VMASKMOV

Scalar	AVX using VMASKMOV

float* pA = A;
float* pB = B;
float* pC = C;
float* pD = D;
float* pE = E;
uint64 len = (uint64)
(miBufferWidth)*sizeof(float);
__asm
{
  mov rax, pA
  mov rbx, pB
  mov rcx, pC
  mov rdx, pD
  mov rsi, pE
  mov r8, len
 	
//xmm8 all zeros
  vxorps xmm8, xmm8, xmm8
  xor r9, r9
	float* pA = A;
	float* pB = B;
	float* pC = C;
	float* pD = D;
	float* pE = E;
	uint64 len = (uint64) (miBufferWidth)*sizeof(float);
	__asm
	{
	  mov rax, pA
	  mov rbx, pB
	  mov rcx, pC
	  mov rdx, pD
	  mov rsi, pE
	  mov r8, len
	//ymm8 all zeros
	vxorps ymm8, ymm8, ymm8
	//ymm9 all ones
	  vcmpps ymm9, ymm8, ymm8, 0
	  xor r9, r9

loop1:
  vmovss xmm1, [rax+r9]
  vcomiss xmm1, xmm8
  jbe a_le
a_gt:
  vmovss xmm4, [rcx+r9]
  jmp mul
a_le:
  vmovss xmm4, [rdx+r9]
mul:
  vmulss xmm4, xmm4, [rsi+r9]
  vmovss [rbx+r9], xmm4
  add r9, 4
  cmp r9, r8
  jl loop1
}
	loop1:
	vmovups ymm1, [rax+r9]
	vcmpps ymm2, ymm8, ymm1, 1
	vmaskmovps ymm4, ymm2, [rcx+r9]
	vxorps ymm2, ymm2, ymm9
	vmaskmovps ymm5, ymm2, [rdx+r9]
	  vorps ymm4, ymm4, ymm5
	  vmulps ymm4, ymm4, [rsi+r9]
	  vmovups [rbx+r9], ymm4
	  add r9, 32
	  cmp r9, r8
	  jl loop1
	}

The performance of the left side of Example 15-14[1] is sensitive to branch mis-predictions and can be an order of magnitude slower than the VMASKMOV example which has no data-dependent branches.

15.10 MIXING INTEGER AND FLOATING-POINT CODE

Integer SIMD functionalities in Intel AVX instructions are limited to 128-bit. There are some algorithm that uses mixed integer SIMD and floating-point SIMD instructions. Therefore, porting such legacy 128-bit code into 256-bit AVX code requires special attention.

For example, PALINGR (Packed Align Right) is an integer SIMD instruction that is useful arranging data elements for integer and floating-point code. But VPALINGR instruction does not have a corresponding 256-bit instruction in AVX.

There are three approaches to consider when porting legacy code consisting of mostly floating-point with some integer operations into 256-bit AVX code:

- Locate a 256-bit AVX alternative to replace the critical128-bit Integer SIMD instructions if such an AVX instructions exist. This is more likely to be true with integer SIMD instruction that rearranges data elements.
- Mix 128-bit AVX and 256-bit AVX instructions[2].
- Use Intel AVX2 instructions.

The performance gain from these two approaches may vary. Where possible, use method (1), since this method utilizes the full 256-bit vector width.

In case the code is mostly integer, convert the code from 128-bit SSE to 128 bit AVX instructions and gain from the Non destructive Source (NDS) feature.

Example 15-15. Three-Tap Filter in C Code

```
for(int i = 0; i < len -2; i++)
{
  pOut[i] = A[i]*coeff[0]+A[i+1]*coeff[1]+A[i+2]*coeff[2];
}
```

Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD

```
    xor ebx, ebx
  mov  rcx, len
  mov  rdi, inPtr
  mov  rsi, outPtr
  mov  r15, coeffs
  movss   xmm2, [r15]    // load coeff 0
  shufps  xmm2, xmm2, 0 // broadcast coeff 0
  movss   xmm1, [r15+4] // load coeff 1
  shufps  xmm1, xmm1, 0 // broadcast coeff 1
  movss   xmm0, [r15+8] // coeff 2
  shufps  xmm0, xmm0, 0// broadcast coeff 2
  movaps xmm5, [rdi] // xmm5={A[n+3],A[n+2],A[n+1],A[n]}
```

1. See Example 14 in GitHub.
2. See Example 16 in GitHub.

Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD (Contd.)

```
loop_start:
  movaps xmm6, [rdi+16] // xmm6={A[n+7],A[n+6],A[n+5],A[n+4]}
  movaps xmm7, xmm6
  movaps xmm8, xmm6
  add rdi, 16          // inPtr+=16
  add rbx, 4           // loop counter
  palignr xmm7, xmm5, 4 // xmm7={A[n+4],A[n+3],A[n+2],A[n+1]}
  palignr xmm8, xmm5, 8 // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
  mulps   xmm5, xmm2   //xmm5={C0*A[n+3],C0*A[n+2],C0*A[n+1], C0*A[n]}

  mulps   xmm7, xmm1   // xmm7={C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
  mulps   xmm8, xmm0   // xmm8={C2*A[n+5],C2*A[n+4] C2*A[n+3],C2*A[n+2]}
  addps   xmm7 ,xmm5
  addps   xmm7, xmm8
  movaps [rsi], xmm7
    movapsxmm5, xmm6
  add rsi, 16       // outPtr+=16
  cmp rbx, rcx
  jl  loop_start
```

Example 15-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS

```
  xorebx, ebx
  mov  rcx, len
  mov  rdi, inPtr
  mov  rsi, outPtr
  mov  r15, coeffs
  vbroadcastss ymm2, [r15]   // load and broadcast coeff 0
  vbroadcastss ymm1, [r15+4] // load and broadcast coeff 1
  vbroadcastss ymm0, [r15+8] // load and broadcast coeff 2


loop_start:
  vmovapsymm5, [rdi]                 // Ymm5={A[n+7],A[n+6],A[n+5],A[n+4];
                                     // A[n+3],A[n+2],A[n+1] , A[n]}
  vshufpsymm6, ymm5, [rdi+16], 0x4e// ymm6={A[n+9],A[n+8],A[n+7],A[n+6];
                                     // A[n+5],A[n+4],A[n+3],A[n+2]}
  vshufpsymm7, ymm5, ymm6, 0x99 // ymm7={A[n+8],A[n+7],A[n+6],A[n+5];
                                     // A[n+4],A[n+3],A[n+2],A[n+1]}
  vmulpsymm3, ymm5, ymm2      // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5],C0*A[n+4];
                                     // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
  vmulpsymm9, ymm7, ymm1      // ymm9={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                                     // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}
  vmulpsymm4, ymm6, ymm0      // ymm4={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                                     // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
  vaddps    ymm8, ymm3, ymm4
  vaddps    ymm10, ymm8, ymm9
  vmovaps[rsi], ymm10
  add   rdi, 32       // inPtr+=32
  add   rbx, 8        // loop counter
  add   rsi, 32       // outPtr+=32
  cmp  rbx, rcx
  jl       loop_start
```

Example 15-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code

```
  xor ebx, ebx
  mov  rcx, len
  mov  rdi, inPtr
  mov  rsi, outPtr
  mov  r15, coeffs
  vbroadcastss ymm2, [r15]          // load and broadcast coeff 0
  vbroadcastss ymm1, [r15+4]        // load and broadcast coeff 1
  vbroadcastss ymm0, [r15+8]        // load and broadcast coeff 2
  vmovaps xmm3, [rdi]               // xmm3={A[n+3],A[n+2],A[n+1],A[n]}
loop_start:
  vmovaps xmm4, [rdi+16]            // xmm4={A[n+7],A[n+6],A[n+5],A[n+4]}
  vmovaps xmm5, [rdi+32]           // xmm5={A[n+11], A[n+10],A[n+9],A[n+8]}
  vinsertf128 ymm3, ymm3, xmm4, 1   // ymm3={A[n+7],A[n+6],A[n+5],A[n+4];
                                    // A[n+3], A[n+2],A[n+1],A[n]}
  vpalignr xmm6, xmm4, xmm3, 4    // xmm6={A[n+4],A[n+3],A[n+2],A[n+1]}
  vpalignr xmm7, xmm5, xmm4, 4    // xmm7={A[n+8],A[n+7],A[n+6],A[n+5]}
  vinsertf128 ymm6, ymm6, xmm7, 1 // ymm6={A[n+8],A[n+7],A[n+6],A[n+5];
                                    // A[n+4],A[n+3],A[n+2],A[n+1]}
  vpalignr xmm8, xmm4, xmm3, 8      // xmm8={A[n+5],A[n+4],A[n+3],A[n+2]}
  vpalignr xmm9, xmm5, xmm4, 8    // xmm9={A[n+9],A[n+8],A[n+7],A[n+6]}
  vinsertf128 ymm8, ymm8, xmm9, 1 // ymm8={A[n+9],A[n+8],A[n+7],A[n+6];
                                    // A[n+5],A[n+4],A[n+3],A[n+2]}
  vmulps ymm3, ymm3, ymm2    // ymm3={C0*A[n+7],C0*A[n+6],C0*A[n+5], C0*A[n+4];
                                    // C0*A[n+3],C0*A[n+2],C0*A[n+1],C0*A[n]}
  vmulps ymm6, ymm6, ymm1    // ymm6={C1*A[n+8],C1*A[n+7],C1*A[n+6],C1*A[n+5];
                                    // C1*A[n+4],C1*A[n+3],C1*A[n+2],C1*A[n+1]}

  vmulps ymm8, ymm8, ymm0    // ymm8={C2*A[n+9],C2*A[n+8],C2*A[n+7],C2*A[n+6];
                                    // C2*A[n+5],C2*A[n+4],C2*A[n+3],C2*A[n+2]}
  vaddps ymm3, ymm3, ymm6
  vaddps ymm3, ymm3, ymm8
  vmovaps[rsi], ymm3
  vmovapsxmm3, xmm5
  add  rdi, 32                    // inPtr+=32
  add  rbx, 8                     // loop counter
  add  rsi, 32                    // outPtr+=32
  cmp  rbx, rcx
  jl  loop_start
```

Example 15-17[1] uses 256-bit VSHUFPS to replace the PALIGNR in 128-bit mixed SSE code. This speeds up almost 70% over the 128-bit mixed SSE code of Example 15-16 and slightly ahead of Example 15-18[2].

For code that includes integer instructions and is written with 256-bit Intel AVX instructions, replace the integer instruction with floating-point instructions that have similar functionality and performance. If there is no similar floating-point instruction, consider using a 128-bit Intel AVX instruction to perform the required integer operation.

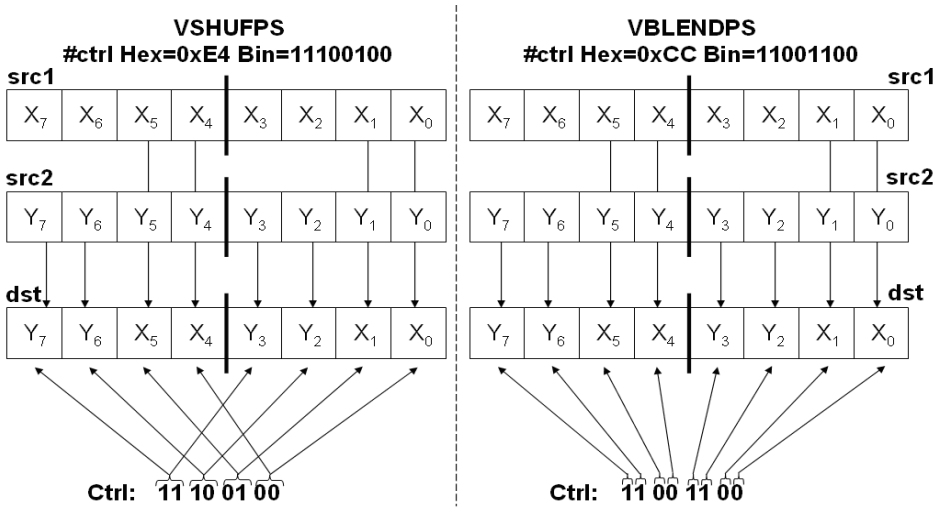
1. See Example 17 in GitHub.
2. See Example 18 in GitHub.

15.11 HANDLING PORT 5 PRESSURE

Port 5 in Sandy Bridge microarchitecture includes shuffle units which frequently become a performance bottleneck. Ice Lake Client microarchitecture has added a restricted, in-lane shuffle unit to port 1 to help reduce some of the pressure. Shuffle operations which can be restructured to operate in-lane will benefit from this unit. Sometimes it is possible to replace shuffle instructions that dispatch only on port 5, with different instructions and improve performance by reducing port 5 pressure.

15.11.1 REPLACE SHUFFLES WITH BLENDS

There are a few cases where shuffles such as VSHUFPS or VPERM2F128 can be replaced by blend instructions. Intel AVX shuffles are executed only on port 5, while blends are also executed on port 0. Therefore, replacing shuffles with blends could reduce port 5 pressure. The following figure shows how a VSHUFPS is implemented using VBLENDPS.



The following example shows two implementations of an 8x8 Matrix transpose. In both cases, the bottleneck is Port 5 pressure. Alternative 1 uses 12 vshufps instructions that are executed only on port 5. Alternative 2 replaces eight of the vshufps instructions with the vblendps instruction which can be executed on Port 0.

Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops loop1: vmovapsymm9, [rcx] vmovapsymm10, [rcx+32] vmovapsymm11, [rcx+64] vmovapsymm12, [rcx+96] vmovapsymm13, [rcx+128] vmovapsymm14, [rcx+160] vmovapsymm15, [rcx+192] vmovapsymm2, [rcx+224] vunpcklpsymm6, ymm9, ymm10 vunpcklpsymm1, ymm11, ymm12 vunpckhpsymm8, ymm9, ymm10 vunpcklpsymm0, ymm13, ymm14 vunpcklpsymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vshufpsymm10, ymm6, ymm3, 0xE4 vshufpsymm6, ymm0, ymm9, 0x4E vunpckhpsymm7, ymm11, ymm12 vshufpsymm11, ymm0, ymm6, 0xE4 vshufpsymm12, ymm3, ymm1, 0xE4 vperm2f128ymm3, ymm10, ymm11, 0x20 vmovaps[rdx], ymm3 vunpckhpsymm5, ymm13, ymm14 vshufpsymm13, ymm6, ymm9, 0xE4 vunpckhpsymm4, ymm15, ymm2 vperm2f128ymm2, ymm12, ymm13, 0x20 vmovaps32[rdx], ymm2 vshufpsymm14, ymm8, ymm7, 0x4E vshufpsymm15, ymm14, ymm7, 0xE4 vshufpsymm7, ymm5, ymm4, 0x4E vshufpsymm8, ymm8, ymm14, 0xE4 vshufpsymm5, ymm5, ymm7, 0xE4 vperm2f128ymm6, ymm8, ymm5, 0x20 vmovaps64[rdx], ymm6	movrcx, inpBuf movrdx, outBuf movr10, NumOfLoops loop1: vmovaps ymm9, [rcx] vmovaps ymm10, [rcx+32] vmovaps ymm11, [rcx+64] vmovaps ymm12, [rcx+96] vmovaps ymm13, [rcx+128] vmovaps ymm14, [rcx+160] vmovaps ymm15, [rcx+192] vmovaps ymm2, [rcx+224] vunpcklps ymm6, ymm9, ymm10 vunpcklps ymm1, ymm11, ymm12 vunpckhps ymm8, ymm9, ymm10 vunpcklps ymm0, ymm13, ymm14 vunpcklps ymm9, ymm15, ymm2 vshufps ymm3, ymm6, ymm1, 0x4E vblendps ymm10, ymm6, ymm3, 0xCC vshufps ymm6, ymm0, ymm9, 0x4E vunpckhps ymm7, ymm11, ymm12 vblendps ymm11, ymm0, ymm6, 0xCC vblendps ymm12, ymm3, ymm1, 0xCC vperm2f128 ymm3, ymm10, ymm11, 0x20 vmovaps [rdx], ymm3 vunpckhps ymm5, ymm13, ymm14 vblendps ymm13, ymm6, ymm9, 0xCC vunpckhps ymm4, ymm15, ymm2 vperm2f128 ymm2, ymm12, ymm13, 0x20 vmovaps 32[rdx], ymm2 vshufps ymm14, ymm8, ymm7, 0x4E vblendps ymm15, ymm14, ymm7, 0xCC vshufps ymm7, ymm5, ymm4, 0x4E vblendps ymm8, ymm8, ymm14, 0xCC vblendps ymm5, ymm5, ymm7, 0xCC vperm2f128 ymm6, ymm8, ymm5, 0x20 vmovaps 64[rdx], ymm6

Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends (Contd.)

256-bit AVX using VSHUFPS	AVX replacing VSHUFPS with VBLENDPS
vshufpsymm4, ymm7, ymm4, 0xE4 vperm2f128ymm7, ymm15, ymm4, 0x20 vmovaps96[rdx], ymm7 vperm2f128ymm1, ymm10, ymm11, 0x31 vperm2f128ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128ymm5, ymm8, ymm5, 0x31 vperm2f128ymm4, ymm15, ymm4, 0x31 vmovaps160[rdx], ymm0 vmovaps192[rdx], ymm5 vmovaps224[rdx], ymm4 decr10 jnzloop1	vblendps ymm4, ymm7, ymm4, 0xCC vperm2f128 ymm7, ymm15, ymm4, 0x20 vmovaps 96[rdx], ymm7 vperm2f128 ymm1, ymm10, ymm11, 0x31 vperm2f128 ymm0, ymm12, ymm13, 0x31 vmovaps 128[rdx], ymm1 vperm2f128 ymm5, ymm8, ymm5, 0x31 vperm2f128 ymm4, ymm15, ymm4, 0x31 vmovaps 160[rdx], ymm0 vmovaps 192[rdx], ymm5 vmovaps 224[rdx], ymm4 dec r10 jnzloop1

In Example 15-19[1], replacing VSHUFPS with VBLENDPS relieved port 5 pressure and can gain almost 40% speedup.

Assembly/Compiler Coding Rule 66. (M impact, M generality) *Use Blend instructions in lieu of shuffle instruction in AVX whenever possible.*

15.11.2 DESIGN ALGORITHM WITH FEWER SHUFFLES

In some cases you can reduce port 5 pressure by changing the algorithm to use less shuffles. Figure 15-5 shows that the transpose moved all the elements in rows 0-4 to the low lanes, and all the elements in rows 4-7 to the high lanes. Therefore, using 256-bit loads in the beginning of the algorithm requires using VPERM2F128 to swap elements between the lanes. The processor executes the VPERM2F128 instruction only on port 5.

Example 15-19 used eight 256-bit loads and eight VPERM2F128 instructions. You can implement the same 8x8 Matrix Transpose using VINSERTF128 instead of the 256-bit loads and the eight VPERM2F128. Using VINSERTF128 from memory is executed in the load ports and on port 0 or 5. The original method required loads that are performed on

1. See Example 19 in GitHub.

the load ports and VPERM2F128 that is only performed on port 5. Therefore redesigning the algorithm to use VINSERTF128 reduces port 5 pressure and improves performance.

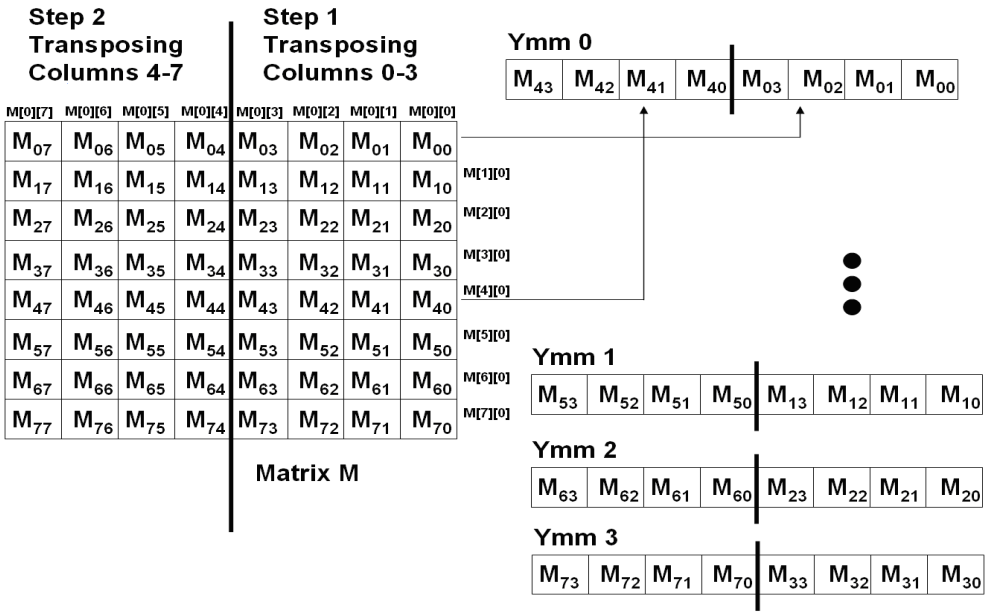


Figure 15-5. Using VPERM2F128 to Swap Elements Between Lanes

Figure 15-6 describes step 1 of the 8x8 matrix transpose with vinsertf128. Step 2 performs the same operations on different columns.

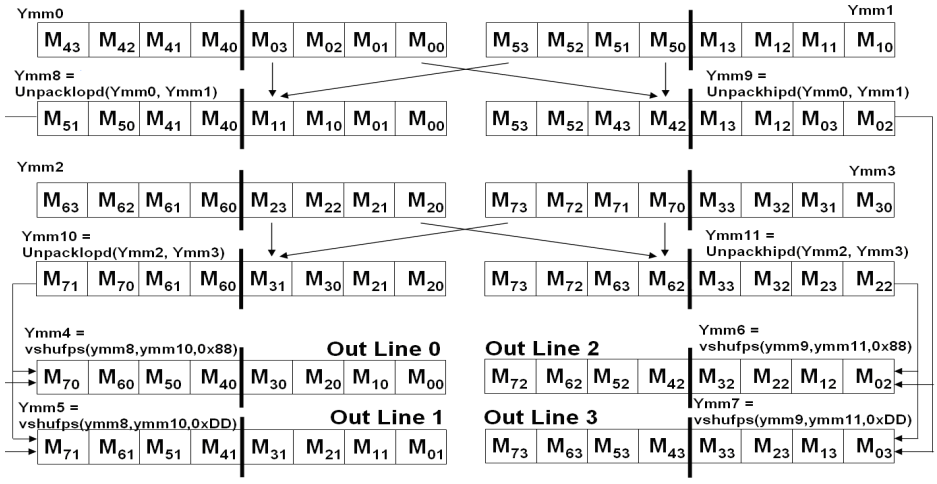


Figure 15-6. Step One of 8 x 8 Matrix Transpose Using VINSERTF128

Example 15-20. 8x8 Matrix Transpose Using VINSERTPS

```
mov     rcx, inpBuf
    mov     rdx, outBuf
    mov     r10, NumOfLoops
loop1:
    vmovaps     xmm0, [rcx]
    vinsertf128 ymm0, ymm0, [rcx + 128], 1
    vmovaps     xmm1, [rcx + 32]
    vinsertf128 ymm1, ymm1, [rcx + 160], 1
vunpcklpd   ymm8, ymm0, ymm1
    vunpckhpd   ymm9, ymm0, ymm1
    vmovaps     xmm2, [rcx+64]
    vinsertf128 ymm2, ymm2, [rcx + 192], 1
    vmovaps     xmm3, [rcx+96]
    vinsertf128 ymm3, ymm3, [rcx + 224], 1
    vunpcklpd   ymm10, ymm2, ymm3
    vunpckhpd   ymm11, ymm2, ymm3
    vshufps     ymm4, ymm8, ymm10, 0x88
    vmovaps     [rdx], ymm4
    vshufps     ymm5, ymm8, ymm10, 0xDD
    vmovaps     [rdx+32], ymm5
    vshufps     ymm6, ymm9, ymm11, 0x88
    vmovaps     [rdx+64], ymm6
    vshufps     ymm7, ymm9, ymm11, 0xDD
    vmovaps     [rdx+96], ymm7
    vmovaps     xmm0, [rcx+16]
    vinsertf128 ymm0, ymm0, [rcx + 144], 1
    vmovaps     xmm1, [rcx + 48]
    vinsertf128 ymm1, ymm1, [rcx + 176], 1

    vunpcklpd   ymm8, ymm0, ymm1
    vunpckhpd   ymm9, ymm0, ymm1

    vmovaps     xmm2, [rcx+80]
    vinsertf128 ymm2, ymm2, [rcx + 208], 1
    vmovaps     xmm3, [rcx+112]
    vinsertf128 ymm3, ymm3, [rcx + 240], 1

    vunpcklpd   ymm10, ymm2, ymm3
    vunpckhpd   ymm11, ymm2, ymm3
    vshufps     ymm4, ymm8, ymm10, 0x88
    vmovaps     [rdx+128], ymm4
    vshufps     ymm5, ymm8, ymm10, 0xDD
    vmovaps     [rdx+160], ymm5
    vshufps     ymm6, ymm9, ymm11, 0x88
    vmovaps     [rdx+192], ymm6
    vshufps     ymm7, ymm9, ymm11, 0xDD
```

Example 15-20. 8x8 Matrix Transpose Using VINSERTPS (Contd.)

```
vmovaps     [rdx+224], ymm7
dec         r10
jnz         loop1
```

In Example 15-20[1], this reduced port 5 pressure further than the combination of VSHUFPS with VBLENDPS in Example 15-19. It can gain 70% speedup relative to relying on VSHUFPS alone in Example 15-19.

15.11.3 PERFORM BASIC SHUFFLES ON LOAD PORTS

Some shuffles can be executed in the load ports (ports 2, 3) if the source is from memory. The following example shows how moving some shuffles (vmovsldup/vmovshdup) from Port 5 to the load ports improves performance significantly.

Figure describes an Intel AVX implementation of the complex multiply algorithm with vmovsldup/vmovshdup on the load ports.

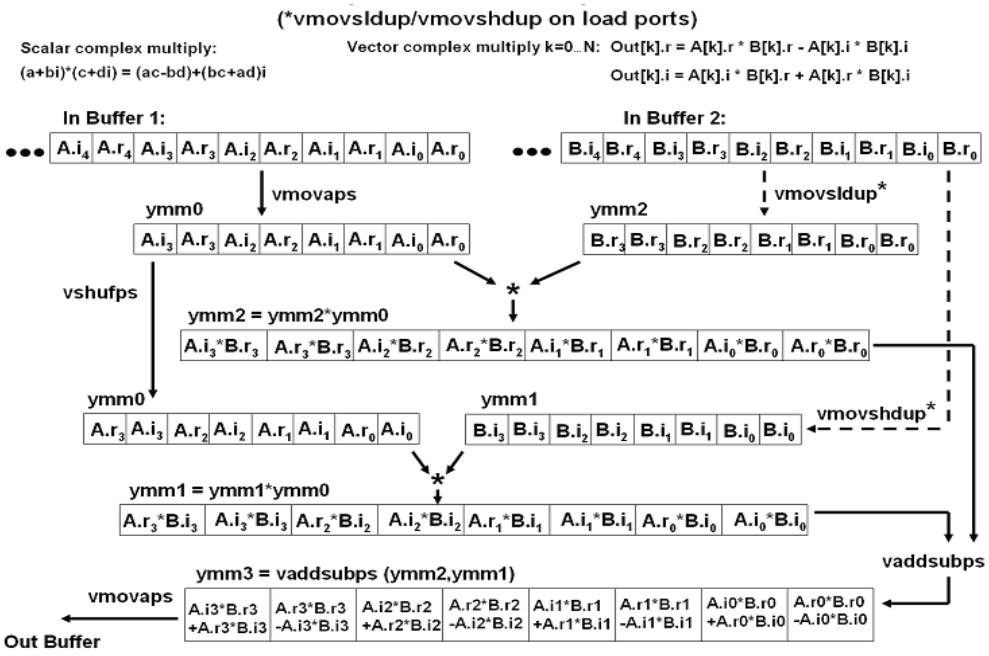


Figure 15-7. Intel AVX Implementation of the Complex Multiply Algorithm

Example 15-21[2] includes two versions of the complex multiply. Both versions are unrolled twice.

Alternative 1 shuffles all the data in registers. Alternative 2 shuffles data while it is loaded from memory.

1. See Example 20 in GitHub.
2. See Example 21 in GitHub.

Example 15-21. Port 5 versus Load Port Shuffles

Shuffles data in registers	Shuffling loaded data

mov rax, inPtr1
mov rbx, inPtr2
mov rdx, outPtr
mov r8, len
xor rcx, rcx
 	
loop1:
vmovaps ymm0, [rax +8*rcx]
vmovaps ymm4, [rax +8*rcx +32]
vmovaps ymm3, [rbx +8*rcx]
vmovsldupymm2, ymm3
vmulps ymm2, ymm2, ymm0
vshufps ymm0, ymm0, ymm0, 177
vmovshdupymm1, ymm3
vmulps ymm1, ymm1, ymm0
vmovaps ymm7, [rbx +8*rcx +32]
vmovsldupymm6, ymm7
vmulps ymm6, ymm6, ymm4
vaddsubpsymm2, ymm2, ymm1
vmovshdupymm5, ymm7
vmovaps [rdx+8*rcx], ymm2
vshufps ymm4, ymm4, ymm4, 177
vmulps ymm5, ymm5, ymm4
vaddsubpsymm6, ymm6, ymm5
vmovaps [rdx+8*rcx+32], ymm6
 	
add rcx, 8
cmp rcx, r8
jl loop1
	mov rax, inPtr1
	mov rbx, inPtr2
	mov rdx, outPtr
	mov r8, len
	xor rcx, rcx
	
	loop1:
	vmovaps ymm0, [rax +8*rcx]
	vmovaps ymm4, [rax +8*rcx +32]
	
	vmovsldupymm2, [rbx +8*rcx]
	vmulps ymm2, ymm2, ymm0
	vshufps ymm0, ymm0, ymm0, 177
	vmovshdupymm1, [rbx +8*rcx]
	vmulps ymm1, ymm1, ymm0
	vmovsldupymm6, [rbx +8*rcx +32]
	vmulps ymm6, ymm6, ymm4
	vaddsubpsymm3, ymm2, ymm1
	vmovshdupymm5, [rbx +8*rcx +32]
	vmovaps [rdx +8*rcx], ymm3
	vshufps ymm4, ymm4, ymm4, 177
	vmulps ymm5, ymm5, ymm4
	vaddsubpsymm7, ymm6, ymm5
	vmovaps [rdx +8*rcx +32], ymm7
	
	add rcx, 8
	cmp rcx, r8
	jl loop1

15.12 DIVIDE AND SQUARE ROOT OPERATIONS

In Intel microarchitectures prior to Skylake, the SSE divide and square root instructions DIVPS and SQRTPS have a latency of 14 cycles (or the neighborhood) and they are not pipelined. This means that the throughput of these instructions is one in every fourteen cycles. The 256-bit Intel AVX instructions VDIVPS and VSQRTPS execute with 128-bit data path and have a latency of twenty-eight cycles and they are not pipelined as well. Therefore, the performance of the Intel SSE divide and square root instructions is similar to the Intel AVX 256-bit instructions on Sandy Bridge microarchitecture.

With the Skylake microarchitecture, 256-bit and 128-bit version of (V)DIVPS/(V)SQRTPS have the same latency because the 256-bit version can execute with a 256-bit data path. The latency is improved and is pipelined to execute with significantly improved throughput. See the 3rd Generation Intel® Xeon® Scalable Processor Family (based on Ice Lake microarchitecture) Instruction Throughput and Latency.

In microarchitectures that provide DIVPS/SQRTPS with high latency and low throughput, it is possible to speed up single-precision divide and square root calculations using the (V)RSQRTPS and (V)RCPPS instructions. For example, with 128-bit RCPPS/RSQRTPS at five-cycle latency and one-cycle throughput or with 256-bit implementation of these

instructions at seven-cycle latency and two-cycle throughput, a single Newton-Raphson iteration or Taylor approximation can achieve almost the same precision as the (V)DIVPS and (V)SQRTPS instructions. See Intel® Architecture Instruction Set Extensions Programming Referencefor more information on these instructions.

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

With the Skylake microarchitecture, choosing between approximate reciprocal instruction alternative versus DIVPS/SQRTPS for optimal performance of simple algebraic computations depend on several factors. Table 15-5 shows several algebraic formula the throughput comparison of implementations of different numeric accuracy tolerances. In each row, 24-bit accurate implementations are IEEE-compliant and using the respective instructions of 128-bit or 256-bit ISA. The columns of 22-bit and 11-bit accurate implementations are using approximate reciprocal instructions of the respective instruction set.

Table 15-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture

Algorithm	Instruction Type	24-bit Accurate	22-bit Accurate	11-bit Accurate
Z = X/Y	SSE	1X	0.9X	1.3X
	256-bit AVX	1X	1.5X	2.6X
$Z = X^{0.5}$	SSE	1X	0.7X	2X
	256-bit AVX	1X	1.4X	3.4X
$Z = X^{-0.5}$	SSE	1X	1.7X	4.3X
	256-bit AVX	1X	3X	7.7X
$Z = (X*Y + Y*Y)^{0.5}$	SSE	1X	0.75X	0.85X
	256-bit AVX	1X	1.1X	1.6X
Z = (X+2Y+3)/(Z-2Y-3)	SSE	1X	0.85X	1X
	256-bit AVX	1X	0.8X	1X

If targeting processors based on the Skylake microarchitecture, Table 15-5 can be summarized as:

- For 256- bit AVX code, Newton-Raphson approximation can be beneficial on Skylake microarchitecture when the algorithm contains only operations executed on the divide unit. However, when single precision divide or square root operations are part of a longer computation, the lower latency of the DIVPS or SQRTPS instructions can lead to better overall performance.
- For SSE or 128-bit AVX implementation, consider use of approximation for divide and square root instructions only for algorithms that do not require precision higher than 11-bit or algorithms that contain multiple operations executed on the divide unit.

Table 15-6 summarizes recommended calculation methods of divisions or square root when using single-precision instructions, based on the desired accuracy level across recent generations of Intel microarchitectures.

Table 15-6. Single-Precision Divide and Square Root Alternatives

Operation	Accuracy Tolerance	Recommendation
Divide	24 bits (IEEE)	DIVPS
	~ 22 bits	Skylake: Consult Table 15-5 Prior uarch: RCPPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RCPPS + MULPS
Reciprocal square root	24 bits (IEEE)	SQRTPS + DIVPS
	~ 22 bits	RSQRTPS + 1 Newton-Raphson Iteration
	~ 11 bits	RSQRTPS
Square root	24 bits (IEEE)	SQRTPS
	~ 22 bits	Skylake: Consult Table 15-5 Prior uarch: RSQRTPS + 1 Newton-Raphson Iteration + MULPS
	~ 11 bits	RSQRTPS + RCPPS

15.12.1 SINGLE-PRECISION DIVIDE

To compute:

Z[i]=A[i]/B[i]

- On a large vector of single-precision numbers, Z[i] can be calculated by a divide operation, or by multiplying 1/B[i] by A[i].
- Denoting B[i] by N, it is possible to calculate 1/N using the (V)RCPPS instruction, achieving approximately 11-bit precision.

For better accuracy you can use the one Newton-Raphson iteration:

X_(0) ~= 1/N ; Initial estimation, rcp(N)

X_(0) = 1/N*(1-E)

E=1-N*X_0 ; E ~= 2^(-11)

X_1=X_0*(1+E)=1/N*(1-E^2) ; E^2 ~= 2^(-22)

X_1=X_0*(1+1-N*X_0)= 2 *X_0 - N*X_0^2

X_1 is an approximation of 1/N with approximately 22-bit precision.

Examples may be seen in GitHub:

- Example 22.
- Example 23.
- Example 24.

Example 15-22. Divide Using DIVPS for 24-bit Accuracy

SSE code using DIVPS	Using VDIVPS

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
divps xmm0, xmm1
movups [rcx+rdx*1], xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx*1]
	vmovups ymm1, [rbx+rdx*1]
	vdivps ymm0, ymm0, ymm1
	vmovups [rcx+rdx*1], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

Example 15-23. Divide Using RCPPS 11-bit Approximation

SSE code using RCPPS	Using VRCPPS

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
 	
loop1:
movups xmm0,[rax+rdx*1]
movups xmm1,[rbx+rdx*1]
rcpps xmm1,xmm1
mulps xmm0,xmm1
movups [rcx+rdx*1],xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx
	
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm1, ymm1
	vmulps ymm0, ymm0, ymm1
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

Example 15-24. Divide Using RCPPS and Newton-Raphson Iteration

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

mov rax, pIn1
mov rbx, pIn2
mov rcx, pOut
mov rsi, iLen
xor rdx, rdx
	mov rax, pIn1
	mov rbx, pIn2
	mov rcx, pOut
	mov rsi, iLen
	xor rdx, rdx

Example 15-24. Divide Using RCPPS and Newton-Raphson Iteration (Contd.)

RCPPS + MULPS ~ 22 bit accuracy	VRCPPS + VMULPS ~ 22 bit accuracy

loop1:
movups xmm0, [rax+rdx*1]
movups xmm1, [rbx+rdx*1]
rcpps xmm3, xmm1
movaps xmm2, xmm3
addps xmm3, xmm2
mulps xmm2, xmm2
mulps xmm2, xmm1
subps xmm3, xmm2
mulps xmm0, xmm3
movups [rcx+rdx*1], xmm0
add rdx, 16
cmp rdx, rsi
jl loop1
	loop1:
	vmovups ymm0, [rax+rdx]
	vmovups ymm1, [rbx+rdx]
	vrcpps ymm3, ymm1
	
	vaddps ymm2, ymm3, ymm3
	vmulps ymm3, ymm3, ymm3
	vmulps ymm3, ymm3, ymm1
	vsubps ymm2, ymm2, ymm3
	vmulps ymm0, ymm0, ymm2
	vmovups [rcx+rdx], ymm0
	add rdx, 32
	cmp rdx, rsi
	jl loop1

15.12.2 SINGLE-PRECISION RECIPROCAL SQUARE ROOT

To compute Z[i]=1/ (A[i]) ^0.5 on a large vector of single-precision numbers, denoting A[i] by N, it is possible to calculate 1/N using the (V)RSQRTPS instruction.

For better accuracy you can use one Newton-Raphson iteration:

X_0 ~=1/N ; Initial estimation RCP(N)

E=1-N*X_0^2

X_0= (1/N)^0.5 * ((1-E)^0.5) = (1/N)^0.5 * (1-E/2) ; E/2~= 2^(-11)

X_1=X_0*(1+E/2) ~= (1/N)^0.5 * (1-E^2/4) ; E^2/4?2^(-22)

X_1=X_0*(1+1/2-1/2*N*X_0^2)= 1/2*X_0*(3-N*X_0^2)

X1 is an approximation of (1/N)^0.5 with approximately 22-bit precision.

Example 15-25. Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy

Using SQRTPS, DIVPS	Using VSQRTPS, VDIVPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
sqrtps xmm0, xmm1
divps xmm0, xmm1
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vmovups ymm1, [rax+rdx]
	vsqrtps ymm0, ymm1
	vdivps ymm0, ymm0, ymm1
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 15-26. Reciprocal Square Root Using RSQRTPS 11-bit Approximation

SSE code using RSQRTPS	Using VRSQRTPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
rsqrtps xmm0, [rax+rdx]
movups [rbx+rdx], xmm0
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	
	loop1:
	vrsqrtps ymm0, [rax+rdx]
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 15-27. Reciprocal Square Root Using RSQRTPS and Newton-Raphson Iteration

RSQRTPS + MULPS ~ 22 bit accuracy	VRSQRTPS + VMULPS ~ 22 bit accuracy

__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};
__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};
__asm
{
mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
movups xmm3, [three]
movups xmm4, [minus_half]
 	
loop1:
movups xmm5, [rax+rdx]
rsqrtps xmm0, xmm5
movaps xmm2, xmm0
mulps xmm0, xmm0
mulps xmm0, xmm5
subps xmm0, xmm3
mulps xmm0, xmm2
mulps xmm0, xmm4
movups [rbx+rdx], xmm0
 	
add rdx, 16
cmp rdx, rcx
jl loop1
}
	__declspec(align(32)) float half[8] = {0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};
	__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
	__asm
	{
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vmovups ymm3, [three]
	vmovups ymm4, [half]
	
	loop1:
	vmovups ymm5, [rax+rdx]
	vrsqrtps ymm0, ymm5
	
	vmulps ymm2, ymm0, ymm0
	vmulps ymm2, ymm2, ymm5
	vsubps ymm2, ymm3, ymm2
	vmulps ymm0, ymm0, ymm2
	vmulps ymm0, ymm0, ymm4
	
	vmovups [rbx+rdx], ymm0
	add rdx, 32
	cmp rdx, rcx
	jl loop1
	}

Detailed Examples may be seen in GitHub:

- Example 25.
- Example 26.
- Example 27.

15.12.3 SINGLE-PRECISION SQUARE ROOT

To compute Z[i]= (A[i])^0.5 on a large vector of single-precision numbers, denoting A[i] by N, the approximation for N^0.5 is N multiplied by (1/N)^0.5 , where the approximation for (1/N)^0.5 is described in the previous section.

To get approximately 22-bit precision of N^0.5, use the following calculation:

N^0.5 = X_1*N = 1/2*N*X_0*(3-N*X_0^2)

Example 15-28. Square Root Using SQRTPS for 24-bit Accuracy

Using SQRTPS	Using VSQRTPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
sqrtps xmm1, xmm1
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	loop1:
	vmovups ymm1, [rax+rdx]
	vsqrtps ymm1,ymm1
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 15-29. Square Root Using RSQRTPS 11-bit Approximation

SSE code using RSQRTPS	Using VRSQRTPS

mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
loop1:
movups xmm1, [rax+rdx]
xorps xmm8, xmm8
cmpneqps xmm8, xmm1
rsqrtps xmm1, xmm1
rcpps xmm1, xmm1
andps xmm1, xmm8
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	vxorps ymm8, ymm8, ymm8
	loop1:
	vmovups ymm1, [rax+rdx]
	vcmpneqps ymm9, ymm8, ymm1
	vrsqrtps ymm1, ymm1
	vrcpps ymm1, ymm1
	vandps ymm1, ymm1, ymm9
	vmovups [rbx+rdx], ymm1
	add rdx, 32
	cmp rdx, rcx
	jl loop1

Example 15-30. Square Root Using RSQRTPS and One Taylor Series Expansion

RSQRTPS + Taylor ~ 22 bit accuracy	VRSQRTPS + Taylor ~ 22 bit accuracy

__declspec(align(16)) float minus_half[4] = {-0.5, -0.5, -0.5, -0.5};
 	
__declspec(align(16)) float three[4] = {3.0, 3.0, 3.0, 3.0};
 	
__asm
{
mov rax, pIn
mov rbx, pOut
mov rcx, iLen
xor rdx, rdx
 	
movups xmm6, [three]
movups xmm7, [minus_half]
loop1:
movups xmm3, [rax+rdx]
rsqrtps xmm1, xmm3
xorps xmm8, xmm8
cmpneqps xmm8, xmm3
andps xmm1, xmm8
movaps xmm4, xmm1
mulps xmm1, xmm3
movaps xmm5, xmm1
mulps xmm1, xmm4
subps xmm1, xmm6
mulps xmm1, xmm5
 	
mulps xmm1, xmm7
movups [rbx+rdx], xmm1
add rdx, 16
cmp rdx, rcx
jl loop1
}
	__declspec(align(32)) float three[8] = {3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0};
	
	__declspec(align(32)) float minus_half[8] = {-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5};
	
	__asm
	{
	mov rax, pIn
	mov rbx, pOut
	mov rcx, iLen
	xor rdx, rdx
	
	vmovups ymm6, [three]
	vmovups ymm7, [minus_half]
	vxorps ymm8, ymm8, ymm8
	loop1:
	vmovups ymm3, [rax+rdx]
	vrsqrtps ymm4, ymm3
	vcmpneqps ymm9, ymm8, ymm3
	vandps ymm4, ymm4, ymm9
	vmulps ymm1, ymm4, ymm3
	vmulps ymm2, ymm1, ymm4
	vsubps ymm2, ymm2, ymm6
	vmulps ymm1, ymm1, ymm2
	vmulps ymm1, ymm1, ymm7
	vmovups [rbx+rdx], ymm1
	
	add rdx, 32
	cmp rdx, rcx
	jl loop1
	}

Detailed Examples may be seen in GitHub:

- Example 28.
- Example 29.
- Example 30.

15.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE

This section shows the transformation of SSE implementation of Array Sub Sum algorithm to Intel AVX implementation.

The Array Sub Sum algorithm is:

$Y_{[i]}$ = Sum of k from 0 to i ($X_{[k]}$) = $X_{[0]} + X_{[1]} + .. + X_{[i]}$

Figure 15-8 describes the SSE implementation.

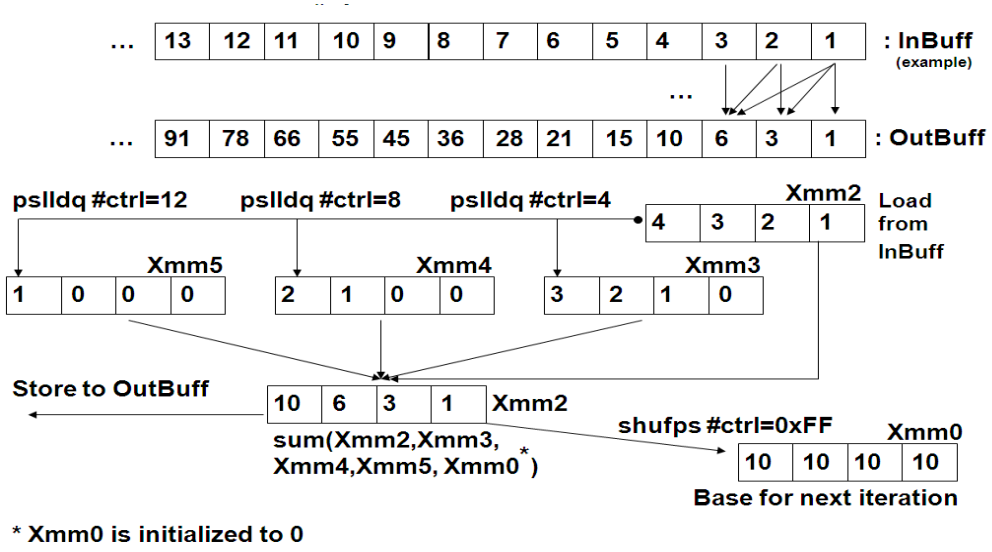


Figure 15-8. SSE Implementation Description

The figure below describes the Intel AVX implementation of the Array Sub Sums algorithm. The PSLLDQ is an integer SIMD instruction which does not have an AVX equivalent. It is replaced by VSHUFPS.

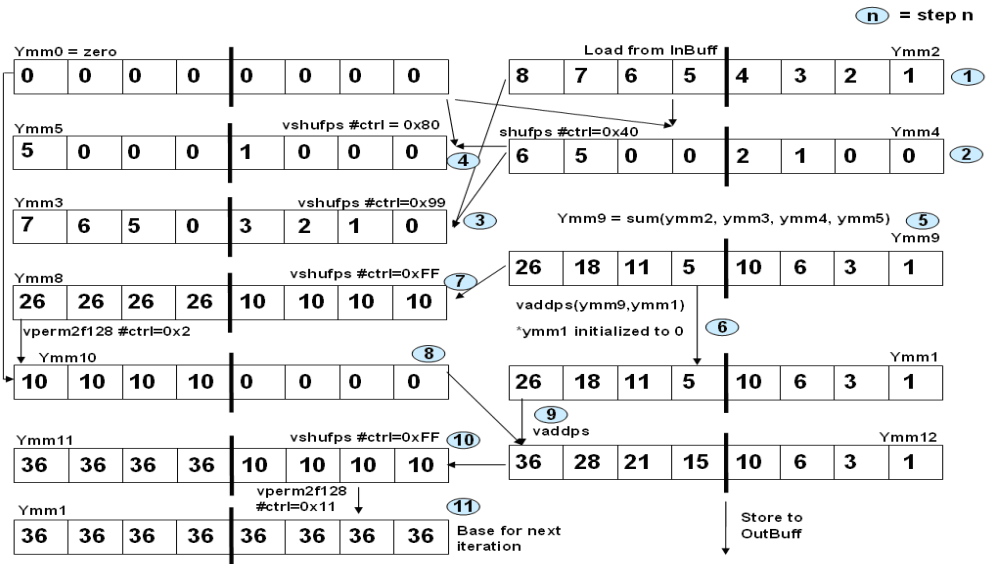


Figure 15-9. Intel® AVX Implementation of the Array Sub Sums Algorithm

Example 15-31. Array Sub Sums Algorithm

SSE code	AVX code

mov rax, InBuff
mov rbx, OutBuff
mov rdx, len
xor rcx, rcx
xorps xmm0, xmm0
 	
loop1:
movapsxmm2, [rax+4*rcx]
movapsxmm3, xmm2
movapsxmm4, xmm2
movapsymm5, ymm2
pslldqxmm3, 4
pslldqxmm4, 8
pslldqxmm5, 12
addps xmm2, xmm3
addps xmm4, xmm5
addps ymm2, xmm4
addps xmm2, xmm0
movapsxmm0, ymm2
shufpsxmm0, xmm2, 0xFF
movaps[rbx+4*rcx], xmm2
add rcx, 4
cmp rcx, rdx
jl loop1
	mov rax, InBuff
	mov rbx, OutBuff
	mov rdx, len
	xor rcx, rcx
	vxorps ymm0, ymm0, ymm0
	vxorps ymm1, ymm1, ymm1
	loop1:
	vmovapsymm2, [rax+4*rcx]
	vshufps ymm4, ymm0, ymm2, 0x40
	vshufps ymm3, ymm4, ymm2, 0x99
	vshufps ymm5, ymm0, ymm4, 0x80
	vaddps ymm6, ymm2, ymm3
	vaddps ymm7, ymm4, ymm5
	vaddps ymm9, ymm6, ymm7
	vaddps ymm1, ymm9, ymm1
	vshufps ymm8, ymm9, ymm9, 0xff
	vperm2f128 ymm10, ymm8, ymm0, 0x2
	vaddps ymm12, ymm1, ymm10
	vshufps ymm11, ymm12, ymm12, 0xff
	vperm2f128 ymm1, ymm11, ymm11, 0x11
	vmovaps [rbx+4*rcx], ymm12
	add rcx, 8
	cmp rcx, rdx
	jl loop1

Example 15-31[1] shows SSE implementation of array sub sum and AVX implementation. The AVX code is about 40% faster, though not on microarchitectures where there are more compute than shuffle ports.

15.14 HALF-PRECISION FLOATING-POINT CONVERSIONS

In applications that use floating-point and require only the dynamic range and precision offered by the 16-bit floating-point format, storing persistent floating-point data encoded in sixteen bits has strong advantages in memory footprint and bandwidth conservation. These situations are encountered in some graphics and imaging workloads.

The encoding format of half-precision floating-point numbers can be found in Chapter 4, "Data Types" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Instructions to convert between packed, half-precision floating-point numbers and packed single-precision floating-point numbers is described in Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 and in the reference pages of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

To perform computations on half precision floating-point data, packed 16-bit FP data elements must be converted to single precision format first, and the single-precision results converted back to half precision format, if necessary. These conversions of 8 data elements using 256-bit instructions are very fast and handle the special cases of denormal numbers, infinity, zero and NaNs properly.

15.14.1 PACKED SINGLE-PRECISION TO HALF-PRECISION CONVERSION

To convert the data in single precision floating-point format to half precision format, without special hardware support like VCVTPS2PH, a programmer needs to do the following:

- Correct exponent bias to permitted range for each data element.
- Shift and round the significand of each data element.
- Copy the sign bit to bit 15 of each element.
- Take care of numbers outside the half precision range.
- Pack each data element to a register of half size.

Example 15-32[2] compares two implementations of floating-point conversion from single precision to half precision. The code on the left uses packed integer shift instructions that is limited to 128-bit SIMD instruction set. The code on right is unrolled twice and uses the VCVTPS2PH instruction.

Example 15-32. Single-Precision to Half-Precision Conversion

AVX-128 code	VCVTPS2PH code

__asm {
mov rax, pIn
mov rbx, pOut
mov rcx, bufferSize
add rcx, rax
	__asm {
	mov rax, pIn
	mov rbx, pOut
	mov rcx, bufferSize
	add rcx, rax

1. See Example 31 in GitHub.
2. See Example 32 in GitHub.

Example 15-32. Single-Precision to Half-Precision Conversion (Contd.)

AVX-128 code	VCVTPS2PH code

vmovdqu xmm0,SignMask16
vmovdqu xmm1,ExpBiasFixAndRound
vmovdqu xmm4,SignMaskNot32
vmovdqu xmm5,MaxConvertibleFloat
vmovdqu xmm6,MinFloat
loop:
vmovdqu xmm2, [rax]
vmovdqu xmm3, [rax+16]
vpaddd xmm7, xmm2, xmm1
vpaddd xmm9, xmm3, xmm1
vpand xmm7, xmm7, xmm4
vpand xmm9, xmm9, xmm4
add rax, 32
vminps xmm7, xmm7, xmm5
vminps xmm9, xmm9, xmm5
vpcmpgtd xmm8, xmm7, xmm6
vpcmpgtd xmm10, xmm9, xmm6
vpand xmm7, xmm8, xmm7
vpand xmm9, xmm10, xmm9
vpackssdw xmm2, xmm3, xmm2
vpsrad xmm7, xmm7, 13
vpsrad xmm8, xmm9, 13
vpand xmm2, xmm2, xmm0
vpackssdw xmm3, xmm7, xmm9
vpaddw xmm3, xmm3, xmm2
vmovdqu [rbx], xmm3
add rbx, 16
cmp rax, rcx
jl loop
	loop:
	vmovups ymm0,[rax]
	vmovups ymm1,[rax+32]
	add rax, 64
	vcvtps2ph [rbx],ymm0, roundingCtrl
	vcvtps2ph [rbx+16],ymm1,roundingCtrl
	add rbx, 32
	cmp rax, rcx
	jl loop

The code using VCVTPS2PH is approximately four times faster than the AVX-128 sequence. Although it is possible to load 8 data elements at once with 256-bit AVX, most of the per-element conversion operations require packed integer instructions which do not have 256-bit extensions yet. Using VCVTPS2PH is not only faster but also provides handling of special cases that do not encode to normal half-precision floating-point values.

15.14.2 PACKED HALF-PRECISION TO SINGLE-PRECISION CONVERSION

Example 15-33[1] compares two implementations using AVX-128 code and with VCVTPH2PS.

Conversion from half precision to single precision floating-point format is easier to implement, yet using VCVTPH2PS instruction performs about 2.5 times faster than the alternative AVX-128 code.

Example 15-33. Half-Precision to Single-Precision Conversion

AVX-128 code	VCVTPS2PH code
__asm { mov rax, pIn	__asm { mov rax, pIn

mov rbx, pOut
mov rcx, bufferSize
add rcx, rax
vmovdqu xmm0,SignMask16
vmovdqu xmm1,ExpBiasFix16
vmovdqu xmm2,ExpMaskMarker
loop:
vmovdqu xmm3, [rax]
add rax, 16
vpandn xmm4, xmm0, xmm3
vpand xmm5, xmm3, xmm0
vpsrlw xmm4, xmm4, 3
vpaddw xmm6, xmm4, xmm1
vpcmpgtw xmm7, xmm6, xmm2
vpand xmm6, xmm6, xmm7
vpand xmm8, xmm3, xmm7
vpor xmm6, xmm6, xmm5
vpsllw xmm8, xmm8, 13
vpunpcklwd xmm3, xmm8, xmm6
vpunpckhwd xmm4, xmm8, xmm6
vmovdqu [rbx], xmm3
vmovdqu [rbx+16], xmm4
add rbx, 32
cmp rax, rcx
jl loop
	mov rbx, pOut
	mov rcx, bufferSize
	add rcx, rax
	loop:
	vcvtph2ps ymm0,[rax]
	vcvtph2ps ymm1,[rax+16]
	add rax, 32
	vmovups [rbx], ymm0
	vmovups [rbx+32], ymm1
	add rbx, 64
	cmp rax, rcx
	jl loop

15.14.3 LOCALITY CONSIDERATION FOR USING HALF-PRECISION FP TO CONSERVE BANDWIDTH

Example 15-32 and Example 15-33 demonstrate the performance advantage of using FP16C instructions when software needs to convert between half-precision and single-precision data. Half-precision FP format is more compact, consumes less bandwidth than single-precision FP format, but sacrifices dynamic range, precision, and incurs conversion overhead if arithmetic computation is required. Whether it is profitable for software to use half-precision data will be highly dependent on locality considerations of the workload.

This section uses an example based on the horizontal median filtering algorithm, “Median3”. The Median3 algorithm calculates the median of every three consecutive elements in a vector:

1. See Example 33 in GitHub.

Y[i] = Median3(X[i], X[i+1], X[i+2])

Where: Y is the output vector, and X is the input vector.

Example 15-34[1] shows two implementations of the Median3 algorithm; one uses single-precision format without conversion, the other uses half-precision format and requires conversion. Alternative one on the left works with single precision format using 256-bit load/store operations, each of which loads/stores eight 32-bit numbers. Alternative two uses 128-bit load/store operations to load/store eight 16-bit numbers in half precision format and VCVTPH2PS/VCVTPS2PH instructions to convert it to/from single precision floating-point format.

Example 15-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision

Single-Precision code w/o Conversion	Half-Precision code w/ Conversion

xor rbx, rbx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr
vmovaps ymm0, [rdi]
	xor rbx, rbx
	mov rcx, len
	mov rdi, inPtr
	mov rsi, outPtr
	vcvtph2ps ymm0, [rdi]

loop:
add rdi, 32
vmovaps ymm6, [rdi]
vperm2f128 ymm1, ymm0, ymm6, 0x21
vshufps ymm3, ymm0, ymm1, 0x4E
vshufps ymm2, ymm0, ymm3, 0x99
vminps ymm5, ymm0, ymm2
vmaxps ymm0, ymm0, ymm2
vminps ymm4, ymm0, ymm3
vmaxps ymm7, ymm4, ymm5
vmovaps ymm0, ymm6
vmovaps [rsi], ymm7
add rsi, 32
add rbx, 8
cmp rbx, rcx
jl loop
	loop:
	add rdi,16
	vcvtph2ps ymm6, [rdi]
	vperm2f128 ymm1, ymm0, ymm6, 0x21
	vshufps ymm3, ymm0, ymm1, 0x4E
	vshufps ymm2, ymm0, ymm3, 0x99
	vminps ymm5, ymm0, ymm2
	vmaxps ymm0, ymm0, ymm2
	vminps ymm4, ymm0, ymm3
	vmaxps ymm7, ymm4, ymm5
	vmovaps ymm0, ymm6
	vcvtps2ph [rsi], ymm7, roundingCtrl
	add rsi, 16
	add rbx, 8
	cmp rbx, rcx
	jl loop

When the locality of the working set resides in memory, using half-precision format with processors based on Ivy Bridge microarchitecture is about 30% faster than single-precision format, despite the conversion overhead. When the locality resides in L3, using half-precision format is still ~15% faster. When the locality resides in L1, using single-precision format is faster because the cache bandwidth of the L1 data cache is much higher than the rest of the cache/memory hierarchy and the overhead of the conversion becomes a performance consideration.

1. See Example 34 in GitHub.

15.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES

FMA instructions perform vectored operations of “a * b + c” on IEEE-754-2008 floating-point values, where the multiplication operations “a * b” are performed with infinite precision, the final results of the addition are rounded to produced the desired precision. Details of FMA rounding behavior and special case handling can be found in section 2.3 of Intel® Architecture Instruction Set Extensions Programming Reference.

FMA instruction can speed up and improve the accuracy of many FP calculations. Haswell microarchitecture implements FMA instructions with execution units on port 0 and port 1 and 256-bit data paths. Dot product, matrix multiplication and polynomial evaluations are expected to benefit from the use of FMA, 256-bit data path and the independent executions on two ports. The peak throughput of FMA from each processor core are 16 single-precision and 8 double-precision results each cycle.

Algorithms designed to use FMA instruction should take into consideration that non-FMA sequence of MULPD/PS and ADDPD/PS likely will produce slightly different results compared to using FMA. For numerical computations involving a convergence criteria, the difference in the precision of intermediate results must be factored into the numeric formalism to avoid surprise in completion time due to rounding issues.

User/Source Coding Rule 29. *Factor in precision and rounding characteristics of FMA instructions when replacing multiply/add operations executing non-FMA instructions.* FMA improves performance when an algorithm is execution-port throughput limited, like DGEMM.

There may be situations where using FMA might not deliver better performance. Consider the vectored operation of “a * b + c * d” and data are ready at the same time:

In the three-instruction sequence of

```
VADDPS ( VMULPS (a,b) , VMULPS (c,b) );
```

VMULPS can be dispatched in the same cycle and execute in parallel, leaving the latency of VADDPS (3 cycle) exposed. With unrolling the exposure of VADDPS latency may be further amortized.

When using the two-instruction sequence of

```
VFMADD213PS ( c, d, VMULPS (a,b) );
```

The latency of FMA (5 cycle) is exposed for producing each vector result.

User/Source Coding Rule 30. *Factor in result-dependency, latency of FP add vs. FMA instructions when replacing FP add operations with FMA instructions.*

15.15.1 OPTIMIZING THROUGHPUT WITH FMA AND FLOATING-POINT ADD/MUL

In the Skylake microarchitecture, there are two pipes of executions supporting FMA, vector FP Multiply, and FP ADD instructions. All three categories of instructions have a latency of 4 cycles and can dispatch to either port 0 or port 1 to execute every cycle.

The arrangement of identical latency and number of pipes allows software to increase the performance of situations where floating-point calculations are limited by the floating-point add operations that follow FP multiplies. Consider a situation of vector operation $A_n = C_1 + C_2 * A_{n-1}$:

Example 15-35. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, ymmword ptr [rbx] // A
vmovups ymm1, ymmword ptr [rcx] // C1
vmovups ymm2, ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
 	
vmovups ymmword ptr[rbx], ymm0 // store A
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, ymmword ptr [rbx] // A
	vmovups ymm1, ymmword ptr [rcx] // C1
	vmovups ymm2, ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	
	vmovups ymmword ptr[rbx], ymm0 // store A

Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

The overall throughput of the code sequence on the LHS is limited by the combined latency of the FP MUL and FP ADD instructions of specific microarchitecture. The overall throughput of the code sequence on the RHS is limited by the throughput of the FMA instruction of the corresponding microarchitecture.

A common situation where the latency of the FP ADD operation dominates performance is the following C code:

for (int 1 = 0; i < arrLenght; i ++) result += arrToSum[i];

Example 15-35[1] shows two implementations with and without unrolling.

Example 15-36. Unrolling to Hide Dependent FP Add Latency

No Unroll	Unroll 8 times

mov eax, arrLength
mov rbx, arrToSum
vmovups ymm0, ymmword ptr [rbx]
sub eax, 8
loop:
add rbx, 32
vaddps ymm0, ymm0, ymmword ptr [rbx]
sub eax, 8
jnz loop
	mov eax, arrLength
	mov rbx, arrToSum
	vmovups ymm0, ymmword ptr [rbx]
	vmovups ymm1, ymmword ptr 32[rbx]
	vmovups ymm2, ymmword ptr 64[rbx]
	vmovups ymm3, ymmword ptr 96[rbx]
	vmovups ymm4, ymmword ptr 128[rbx]
	vmovups ymm5, ymmword ptr 160[rbx]
	vmovups ymm6, ymmword ptr 192[rbx]
	vmovups ymm7, ymmword ptr 224[rbx]

1. See Example 35 GitHub.

Example 15-36. Unrolling to Hide Dependent FP Add Latency (Contd.)

No Unroll	Unroll 8 times

vextractf128 xmm1, ymm0, 1
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0xe
vaddps xmm0, xmm0, xmm1
vpermilps xmm1, xmm0, 0x1
vaddss xmm0, xmm0, xmm1
movss result, xmm0
	sub eax, 64
	loop:
	add rbx, 256
	vaddps ymm0, ymm0, ymmword ptr [rbx]
	vaddps ymm1, ymm1, ymmword ptr 32[rbx]
	vaddps ymm2, ymm2, ymmword ptr 64[rbx]
	vaddps ymm3, ymm3, ymmword ptr 96[rbx]
	vaddps ymm4, ymm4, ymmword ptr 128[rbx]
	vaddps ymm5, ymm5, ymmword ptr 160[rbx]
	vaddps ymm6, ymm6, ymmword ptr 192[rbx]
	vaddps ymm7, ymm7, ymmword ptr 224[rbx]
	sub eax, 64
	jnz loop
	
	vaddps ymm0, ymm0, ymm1
	vaddps ymm2, ymm2, ymm3
	vaddps ymm4, ymm4, ymm5
	vaddps ymm6, ymm6, ymm7
	vaddps ymm0, ymm0, ymm2
	vaddps ymm4, ymm4, ymm6
	vaddps ymm0, ymm0, ymm4
	
	vextractf128 xmm1, ymm0, 1
	vaddps xmm0, xmm0, xmm1
	vpermilps xmm1, xmm0, 0xe
	vaddps xmm0, xmm0, xmm1
	vpermilps xmm1, xmm0, 0x1
	vaddss xmm0, xmm0, xmm1
	movss result, xmm0

Without unrolling (LHS of Example 15-35), the cost of summing every eight array elements is about proportional to the latency of the FP ADD instruction, assuming the working set fit in L1. To use unrolling effectively, the number of unrolled operations should be at least “latency of the critical operation” * “number of pipes”. The performance gain of optimized unrolling versus no unrolling, for a given microarchitecture, can approach “number of pipes” * “Latency of FP ADD”.

User/Source Coding Rule 31. *Consider using unrolling technique for loops containing back-to-back dependent FMA, FP Add or Vector MUL operations, The unrolling factor can be chosen by considering the latency of the critical instruction of the dependency chain and the number of pipes available to execute that instruction.*

15.15.2 OPTIMIZING THROUGHPUT WITH VECTOR SHIFTS

In the Skylake microarchitecture, many common vector shift instructions can dispatch into either port 0 or port 1, compared to only one port in prior generations, see Table 2-14.

A common situation where the latency of the FP ADD operation dominates performance is the following C code, where a, b, and c are integer arrays:

```
for ( int 1 = 0; i < len; i ++) c[i] += 4* a[i] + b[i]/2;
```

Example 15-35 shows two implementations with and without unrolling.

Example 15-37. FP Mul/FP Add Versus FMA

FP Mul/FP Add Sequence	FMA Sequence

mov eax, NumOfIterations
mov rbx, pA
mov rcx, pC1
mov rdx, pC2
vmovups ymm0, ymmword ptr [rbx] // A
vmovups ymm1, ymmword ptr [rcx] // C1
vmovups ymm2, ymmword ptr [rdx] // C2
loop:
vmulps ymm4, ymm0 ,ymm2 // A * C2
vaddps ymm0, ymm1, ymm4
dec eax
jnz loop
 	
vmovups ymmword ptr[rbx], ymm0 // store An
	mov eax, NumOfIterations
	mov rbx, pA
	mov rcx, pC1
	mov rdx, pC2
	vmovups ymm0, ymmword ptr [rbx] // A
	vmovups ymm1, ymmword ptr [rcx] // C1
	vmovups ymm2, ymmword ptr [rdx] // C2
	loop:
	vfmadd132ps ymm0, ymm1, ymm2 // C1 + A * C2
	dec eax
	jnz loop
	
	vmovups ymmword ptr[rbx], ymm0 // store An

Cost per iteration: ~ fp add latency + fp add latency	Cost per iteration: ~ fma latency

15.16 AVX2 OPTIMIZATION GUIDELINES

AVX2 instructions promotes the great majority of 128-bit SIMD integer instructions to operate on 256-bit YMM registers. AVX2 also adds a rich mix of broadcast/permute/variable-shift instructions to accelerate numerical computations. The 256-bit AVX2 instructions are supported by Haswell microarchitecture, which implements 256-bit data path with low latency and high throughput.

Consider an intra-coding 4x4 block image transformation[1] shown in Figure 15-10.

A 128-bit SIMD implementation can perform this transformation by the following technique:

- Convert 8-bit pixels into 16-bit word elements and fetch two 4x4 image block as 4 row vectors.
- The matrix operation 1/128 * (B x R) can be evaluated with row vectors of the image block and column vectors of the right-hand-side coefficient matrix using a sequence of SIMD instructions of PMADDWD, packed shift and blend instructions.
- The two 4x4 word-granular, intermediate result can be re-arranged into column vectors.
- The left-hand-side coefficient matrix in row vectors and the column vectors of the intermediate block can be calculated (using PMADDWD, shift, blend) and written out.

1. C. Yeo, Y. H. Tan, Z. Li and S. Rahardja, “Mode-Dependent Fast Separable KLT for Block-based Intra Coding,” JCTVC-B024, Geneva, Switzerland, Jul 2010

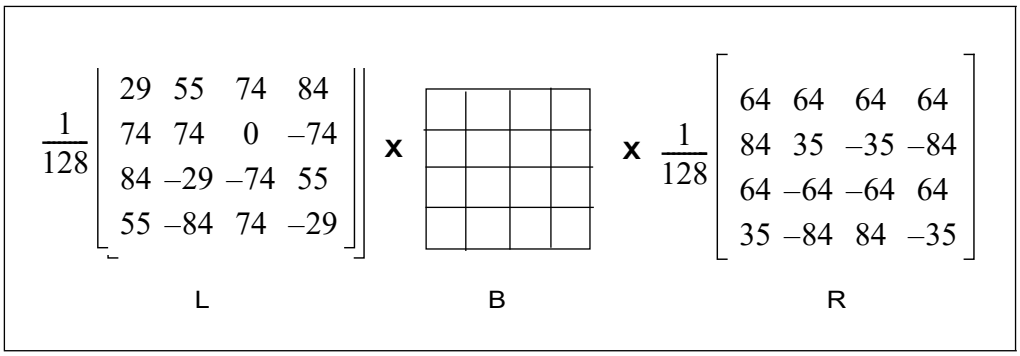


Figure 15-10. 4x4 Image Block Transformation

The same technique can be implemented using AVX2 instructions in a straightforward manner. The AVX2 sequence is illustrated in Example 15-38[1] and Example 15-39.

Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2

```
// b0: input row vector from 4 consecutive 4x4 image block of word pixels
// rmc0-3: columnar vector coefficient of the RHS matrix, repeated 4X for 256-bit
// min32km1: saturation constant vector to cap intermediate pixel to less than or equal to 32767
// w0: output row vector of garbled intermediate matrix, elements within each block are garbled
// e.g Low 128-bit of row 0 in descending order: y07, y05, y06, y04, y03, y01, y02, y00

#define __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0_256,
rmc1_256, rmc2_256, rmc3_256, min32km1)\
  {__m256i  tt0, tt1, tt2, tt3, tttmp;\
  tt0 = _mm256_madd_epi16(b0, (rmc0_256));\
  tt1 = _mm256_madd_epi16(b0, rmc1_256);\
  tt1 = _mm256_hadd_epi32(tt0, tt1);\
  tttmp = _mm256_srai_epi32( tt1, 31);\
  tttmp = _mm256_srli_epi32( tttmp, 25);\
  tt1 = _mm256_add_epi32( tt1, tttmp);\
  tt1 = _mm256_min_epi32(_mm256_srai_epi32( tt1, 7), min32km1);\
  tt1 = _mm256_shuffle_epi32(tt1, 0xd8); \
  tt2 = _mm256_madd_epi16(b0, rmc2_256);\
  tt3 = _mm256_madd_epi16(b0, rmc3_256);\
  tt3 = _mm256_hadd_epi32(tt2, tt3);\
```

1. See Example 38 in GitHub.

Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
   tttmp = _mm256_srai_epi32( tt3, 31);\
   tttmp = _mm256_srli_epi32( tttmp, 25);\
   tt3 = _mm256_add_epi32( tt3, tttmp);\
   tt3 = _mm256_min_epi32( _mm256_srai_epi32(tt3, 7), min32km1);\
   tt3 = _mm256_shuffle_epi32(tt3, 0xd8);\
   w0 = _mm256_blend_epi16(tt1, _mm256_slli_si256( tt3, 2),  0xaa);\
}
// t0-t3: 256-bit input vectors of un-garbled intermediate matrix 1/128 * (B x R)
// lmr_256: 256-bit vector of one row of LHS coefficient, repeated 4X
// min32km1: saturation constant vector to cap final pixel to less than or equal to 32767
// w0; Output row vector of final result in un-garbled order
#define __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr_256, min32km1)\
    {__m256i tb0, tb1, tb2, tb3, tbtmp;
    tb0 = _mm256_madd_epi16( lmr_256, t0);\
    tb1 = _mm256_madd_epi16( lmr_256, t1);\
    tb1 = _mm256_hadd_epi32(tb0, tb1);\
    tbtmp = _mm256_srai_epi32( tb1, 31);\
    tbtmp = _mm256_srli_epi32( tbtmp, 25);\
    tb1 = _mm256_add_epi32( tb1, tbtmp);\
    tb1 = _mm256_min_epi32( _mm256_srai_epi32( tb1, 7), min32km1);\
    tb1 = _mm256_shuffle_epi32(tb1, 0xd8);\
    tb2 = _mm256_madd_epi16( lmr_256, t2);\
    tb3 = _mm256_madd_epi16( lmr_256, t3);\
    tb3 = _mm256_hadd_epi32(tb2, tb3);\
    tbtmp = _mm256_srai_epi32( tb3, 31);\
    tbtmp = _mm256_srli_epi32( tbtmp, 25);\
    tb3 = _mm256_add_epi32( tb3, tbtmp);\
    tb3 = _mm256_min_epi32( _mm256_srai_epi32( tb3, 7), min32km1);\
    tb3 = _mm256_shuffle_epi32(tb3, 0xd8); \
    tb3 = _mm256_slli_si256( tb3, 2);\
    tb3 = _mm256_blend_epi16(tb1, tb3,  0xaa);\
    w0 = _mm256_shuffle_epi8(tb3, _mm256_setr_epi32( 0x5040100, 0x7060302, 0xd0c0908, 0xf0e0b0a, 0x5040100,
0x7060302, 0xd0c0908, 0xf0e0b0a));\
}
```

In Example 15-39[1], matrix multiplication of 1/128 * (B xR) is evaluated first in a four-wide manner by fetching from four consecutive 4x4 image block of word pixels. The first macro shown in Example 15-38 produces an output vector where each intermediate row result is in an garbled sequence between the two middle elements of each 4x4 block. In Example 15-39, undoing the garbled elements and transposing the intermediate row vector into column vectors are implemented using blend primitives instead of shuffle/unpack primitives.

In Haswell microarchitecture, shuffle/pack/unpack primitives rely on the shuffle execution unit dispatched to port 5. In some situations of heavy SIMD sequences, port 5 pressure may become a determining factor in performance.

1. See Example 39 in GitHub.

If 128-bit SIMD code faces port 5 pressure when running on Haswell microarchitecture, porting 128-bit code to use 256-bit AVX2 can improve performance and alleviate port 5 pressure.

Example 15-39. Separable KLT Intra-block Transformation Using AVX2

```
short __declspec(align(16))cst_rmc0[8] = {64, 84, 64, 35, 64, 84, 64, 35};
short __declspec(align(16))cst_rmc1[8] = {64, 35, -64, -84, 64, 35, -64, -84};
short __declspec(align(16))cst_rmc2[8] = {64, -35, -64, 84, 64, -35, -64, 84};
short __declspec(align(16))cst_rmc3[8] = {64, -84, 64, -35, 64, -84, 64, -35};
short __declspec(align(16))cst_lmr0[8] = {29, 55, 74, 84, 29, 55, 74, 84};
short __declspec(align(16))cst_lmr1[8] = {74, 74, 0, -74, 74, 74, 0, -74};
short __declspec(align(16))cst_lmr2[8] = {84, -29, -74, 55, 84, -29, -74, 55};
short __declspec(align(16)) cst_lmr3[8] = {55, -84, 74, -29, 55, -84, 74, -29};

void Klt_256_d(short * Input, short * Output, int iWidth, int iHeight)
{int iX, iY;
__m256i rmc0 = _mm256_broadcastsi128_si256( _mm_loadu_si128((__m128i *) &cst_rmc0[0]));
__m256i rmc1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc1[0]));
__m256i rmc2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc2[0]));
__m256i rmc3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_rmc3[0]));
__m256i lmr0 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr0[0]));
__m256i lmr1 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr1[0]));
__m256i lmr2 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr2[0]));
__m256i lmr3 = _mm256_broadcastsi128_si256(  _mm_loadu_si128((__m128i *)&cst_lmr3[0]));
__m256i min32km1 = _mm256_broadcastd_epi32(_mm_cvtsi32_si128( _mm_setr_epi32( 0x7fff7fff, 0x7fff7fff, 0x7fff7fff,
0x7fff7fff));
__m256i b0, b1, b2, b3, t0, t1, t2, t3;
__m256i w0, w1, w2, w3;
short* pImage = Input;
short* pOutImage = Output;
int hgt = iHeight, wid= iWidth;
// We implement 1/128 * (Mat_L x (1/128 * (Mat_B x Mat_R))) from the inner most parenthesis
for( iY = 0; iY < hgt; iY+=4) {
  for( iX = 0; iX < wid; iX+=16) {
     //load row 0 of 4 consecutive 4x4 matrix of word pixels
     b0 = _mm256_loadu_si256( (__m256i *) (pImage + iY*wid+ iX)) ;
     // multiply row 0 with columnar vectors of the RHS matrix coefficients
     __MyM_KIP_PxRMC_ROW_4x4Wx4(b0, w0, rmc0, rmc1, rmc2, rmc3, min32km1);
// low 128-bit of garbled row 0, from hi->lo: y07, y05, y06, y04, y03, y01, y02, y00
     b1 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+1)*wid+ iX) );
     __MyM_KIP_PxRMC_ROW_4x4Wx4(b1, w1, rmc0, rmc1, rmc2, rmc3, min32km1);
```

```
     // hi->lo y17, y15, y16, y14, y13, y11, y12, y10
     b2 = _mm256_loadu_si256( (__m256i *)  (pImage + (iY+2)*wid+ iX) );
     __MyM_KIP_PxRMC_ROW_4x4Wx4(b2, w2, rmc0, rmc1, rmc2, rmc3, min32km1);
     b3 = _mm256_loadu_si256( (__m256i *) (pImage + (iY+3)*wid+ iX) );
     __MyM_KIP_PxRMC_ROW_4x4Wx4(b3, w3, rmc0, rmc1, rmc2, rmc3, min32km1);
```

Example 15-39. Separable KLT Intra-block Transformation Using AVX2 (Contd.)

```
        // unscramble garbled middle 2 elements of each 4x4 block, then
        // transpose into columnar vectors: t0 has 4 consecutive column 0 or 4 4x4 intermediate
        t0 = _mm256_blend_epi16( w0, _mm256_slli_epi64(w1, 16), 0x22);
        t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w2, 32), 0x44);
        t0 = _mm256_blend_epi16( t0, _mm256_slli_epi64(w3, 48), 0x88);
        t1 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 32), _mm256_srli_epi64(w1, 16), 0x22);
        t1 = _mm256_blend_epi16( t1, w2, 0x44);
        t1 = _mm256_blend_epi16( t1, _mm256_slli_epi64(w3, 16), 0x88); // column 1
        t2 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 16), w1, 0x22);
        t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w2, 16), 0x44);
        t2 = _mm256_blend_epi16( t2, _mm256_slli_epi64(w3, 32), 0x88); // column 2
        t3 = _mm256_blend_epi16( _mm256_srli_epi64(w0, 48), _mm256_srli_epi64(w1, 32), 0x22);
        t3 = _mm256_blend_epi16( t3, _mm256_srli_epi64(w2, 16), 0x44);
        t3 = _mm256_blend_epi16( t3, w3, 0x88);// column 3
        // multiply row 0 of the LHS coefficient with 4 columnar vectors of intermediate blocks
        // final output row are arranged in normal order

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w0, t0, t1, t2, t3, lmr0, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+iY*wid+ iX), w0) ;

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w1, t0, t1, t2, t3, lmr1, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+1)*wid+ iX), w1) ;

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w2, t0, t1, t2, t3, lmr2, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+2)*wid+ iX), w2) ;

        __MyM_KIP_LMRxP_ROW_4x4Wx4(w3, t0, t1, t2, t3, lmr3, min32km1);
        _mm256_store_si256( (__m256i *) (pOutImage+(iY+3)*wid+ iX), w3) ;
    }
}
}
```

Although 128-bit SIMD implementation is not shown here, it can be easily derived.

When running 128-bit SIMD code of this KLT intra-coding transformation on Sandy Bridge microarchitecture, the port 5 pressure are less because there are two shuffle units, and the effective throughput for each 4x4 image block transformation is around fifty cycles. Its speed-up relative to optimized scalar implementation is about 2.5X.

When the 128-bit SIMD code runs on Haswell microarchitecture, micro-ops issued to port 5 account for slightly less than 50% of all micro-ops, compared to about one third on prior microarchitecture, resulting in about 25% performance regression. On the other hand, AVX2 implementation can deliver effective throughput in less than thirty-five cycle per 4x4 block.

15.16.1 MULTI-BUFFERING AND INTEL® AVX2

There are many compute-intensive algorithms (e.g. hashing, encryption, etc.) which operate on a stream of data buffers. Very often, the data stream may be partitioned and treated as multiple independent buffer streams to leverage SIMD instruction sets.

Detailed treatment of hashing several buffers in parallel are included in two papers:

- Simultaneous Hashing of Multiple Messages
- A j-lanes tree hashing mode and j-lanes SHA-256

With Intel AVX2 providing a full compliment of 256-bit SIMD instructions with rich functionality at multiple width granularities for logical and arithmetic operations. Algorithms that had leveraged XMM registers and prior generations of SSE instruction sets can extend those multi-buffering algorithms to use AVX2 on YMM and deliver even higher throughput. Optimized 256-bit Intel AVX2 implementation may deliver up to 1.9X throughput when compared to 128-bit versions.

The image block transformation example discussed in Section 15.16 can be construed also as a multi-buffering implementation of 4x4 blocks. When the performance baseline is switched from a two-shuffle-port microarchitecture (Sandy Bridge) to single-shuffle-port microarchitecture, the 256-bit wide AVX2 provides a speed up of 1.9X relative to 128-bit SIMD implementation.

15.16.2 MODULAR MULTIPLICATION AND AVX2

Modular multiplication of very large integers are often used to implement efficient modular exponentiation operations which are critical in public key cryptography, such as RSA 2048. Library implementation of modular multiplication is often done with MUL/ADC chain sequences. Typically, a MUL instruction can produce a 128-bit intermediate integer output, and add-carry chains must be used at 64-bit intermediate data granularity.

In AVX2, VPMULUDQ/VPADDQ/VPSRLQ/VPSLLQ/VPBROADCASTQ/VPERMQ allow vectorized approach to implement efficient modular multiplication/exponentiation for key lengths corresponding to RSA1024 and RSA2048[1].

The basic heuristic starts with reformulating the large integer input operands in 512/1024 bit exponentiation in redundant representations. For example, a 1024-bit integer can be represented using base 2^29 and 36 "**digits**", where each "**digit**" is less than 2^29. A digit in such redundant representation can be placed in a dword slot of a vector register. Such redundant representation of large integer simplifies the requirement to perform carry-add chains across the hardware granularity of the intermediate results of unsigned integer multiplications.

Each VPMULUDQ in AVX2 using the **digits** from a redundant representation can produce 4 separate 64-bit intermediate result with sufficient headroom (e.g. five most significant bits are 0 excluding sign bit). Then, VPADDQ is sufficient to implement add-carry chain requirement without needing SIMD versions of equivalent of ADC-like instructions. More details are available in the reference cited in paragraph above, including the cost factor of conversion to redundant representation and effective speedup accounting for parallel output bandwidth of VPMULUDQ/VPADDQ chain.

15.16.3 DATA MOVEMENT CONSIDERATIONS

Haswell microarchitecture can support up to two 256-bit loads and one 256-bit store micro-ops dispatched each cycle. Most existing binaries with heavy data-movement operation can benefit from this enhancement and the higher bandwidths of the L1 data cache and L2 without re-compilation, if the binary is already optimized for the prior generation microarchitecture. For example, 256-bit SAXPY computation was limited by the number of load/store

1. For details of modular exponentiation/multiplication and AVX2 implementation in OpenSSL, see: Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures.

ports available in the previous microarchitecture generation; it will benefit immediately on Haswell microarchitecture.

In some situations, there may be some intricate interactions between microarchitectural restrictions on the instruction set that is worth some discussion. We consider two commonly used library functions memcpy() and memset() and the optimal choice to implement them on the new microarchitecture.

With memcpy() on Haswell microarchitecture, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than 20 bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is slower than using 128-bit SIMD technique described in Section 15.16.3.1.

With memcpy() on Ice Lake microarchitecture, using in-lined REP MOVSB to implement memcpy is as fast as a 256-bit AVX implementation for copy lengths that are variable and unknown at compile time. For lengths that are known at compile time, REP MOVSB is almost as good as 256-bit AVX for short strings up to 128 bytes (nine cycles vs three to seven cycles), and better for strings of 2K bytes and longer. For these cases we recommend using inline REP MOVSB. That said, software should still branch away for zero byte copies.

15.16.3.1 SIMD Heuristics to implement Memcpy()

We start with a discussion of the general heuristic to attempt implementing memcpy() with 128-bit SIMD instructions, which revolves around three numeric factors (destination address alignment, source address alignment, bytes to copy) relative to the width of register width of the desired instruction set. The data movement work of memcpy can be separated into the following phases:

- An initial unaligned copy of 16 bytes, allows looping destination address pointer to become 16-byte aligned. Thus subsequent store operations can use as many 16-byte aligned stores.
- The remaining bytes-left-to-copy are decomposed into (a) multiples of unrolled 16-byte copy operations, plus (b) residual count that may include some copy operations of less than 16 bytes. For example, to unroll eight time to amortize loop iteration overhead, the residual count must handle individual cases from 1 to 8x16-1 = 127.
- Inside an 8X16 unrolled main loop, each 16 byte copy operation may need to deal with source pointer address is not aligned to 16-byte boundary and store 16 fresh data to 16B-aligned destination address. When the iterating source pointer is not 16B-aligned, the most efficient technique is a three instruction sequence of:
  - Fetch an 16-byte chunk from an 16-byte-aligned adjusted pointer address and use a portion of this chunk with complementary portion from previous 16-byte-aligned fetch.
  - Use PALIGNR to stitch a portion of the current chunk with the previous chunk.
  - Stored stitched 16-byte fresh data to aligned destination address, and repeat this 3 instruction sequence.

  This 3-instruction technique allows the fetch:store instruction ratio for each 16-byte copy operation to remain at 1:1.

While the above technique (specifically, the main loop dealing with copying thousands of bytes of data) can achieve throughput of approximately 10 bytes per cycle on Sandy Bridge and Ivy Bridge microarchitectures with 128-bit data path for store operations, an attempt to extend this technique to use wider data path will run into the following restrictions:

- To use 256-bit VPALIGNR with its 2X128-bit lane microarchitecture, stitching of two partial chunks of the current 256-bit 32-byte-aligned fetch requires another 256-bit fetch from an address 16-byte offset from the current 32-byte-aligned 256-bit fetch.
  - The fetch:store ratio for each 32-byte copy operation becomes 2:1.
  - The 32-byte-unaligned fetch (although aligned to 16-byte boundary) will experience a cache-line split penalty, once every 64-bytes of copy operation.

The net of this attempt to use 256-bit ISA to take advantage of the 256-bit store data-path microarchitecture was offset by the 4-instruction sequence and cacheline split penalty.

15.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB

It is interesting to compare the alternate approach of using enhanced REP MOVSB to implement memcpy(). In Haswell and Ivy Bridge microarchitectures, REP MOVSB is an optimized, hardware provided, micro-op flow.

On Ivy Bridge microarchitecture, a REP MOVSB implementation of memcpy can achieve throughput at slightly better than the 128-bit SIMD implementation when copying thousands of bytes. However, if the size of copy operation is less than a few hundred bytes, the REP MOVSB approach is less efficient than the explicit residual copy technique described in phase two of Section 15.16.3.1. This is because handling 1-127 residual copy length (via jump table or switch/case, and is done before the main loop) plus one or two 8x16B iterations incurs less branching overhead than the hardware provided micro-op flows. For the grueling implementation details of 128-bit SIMD implementation of memcpy(), one can look up from the archived sources of open source library such as GLibC.

On Haswell microarchitecture, using REP MOVSB to implement memcpy operation for large copy length can take advantage the 256-bit store data path and deliver throughput of more than twenty bytes per cycle. For copy length that are smaller than a few hundred bytes, REP MOVSB approach is still slower than treating the copy length as the residual phase of Section 15.16.3.1.

15.16.3.3 Memset() Implementation Considerations

The interface of Memset() has one address pointer as destination, which simplifies the complexity of managing address alignment scenarios to use 256-bit aligned store instruction. After an initial unaligned store, and adjusting the destination pointer to be 32-byte aligned, the residual phase follows the same consideration as described in Section 15.16.3.1, which may employ a large jump table to handle each residual value scenario with minimal branching, depending on the amount of unrolled 32B-aligned stores. The main loop is a simple YMM register to 32-byte-aligned store operation, which can deliver close to 30 bytes per cycle for lengths more than a thousand byte. The limiting factor here is due to each 256-bit VMOVDQA store consists of a store_address and a store_data micro-op flow. Only port 4 is available to dispatch the store_data micro-op each cycle.

Using REP STOSB to implement memset() has the code size advantage versus a SIMD implementation, like REP MOVSB for memcpy(). On Haswell microarchitecture, a memset() routine implemented using REP STOSB will also benefit the from the 256-bit data path and increased L1 data cache bandwidth to deliver up to 32 bytes per cycle for large count values.

Comparing the performance of memset() implementations using REP STOSB vs. 256-bit AVX2 requires one to consider the pattern of invocation of memset(). The invocation pattern can lead to the necessity of using different performance measurement techniques. There may be side effects affecting the outcome of each measurement technique.

The most common measurement technique that is often used with a simple routine like memset() is to execute memset() inside a loop with a large iteration count, and wrap the invocation of RDTSC before and after the loop.

A slight variation of this measurement technique can apply to measuring memset() invocation patterns of multiple back-to-back calls to memset() with different count values with no other intervening instruction streams executed between calls to memset().

In both of the above memset() invocation scenarios, branch prediction can play a significant role in affecting the measured total cycles for executing the loop. Thus, measuring AVX2-implemented memset() under a large loop to minimize RDTSC overhead can produce a skewed result with the branch predictor being trained by the large loop iteration count.

In more realistic software stacks, the invocation patterns of memset() will likely have the characteristics that:

- There are intervening instruction streams being executed between invocations of memset(), the state of branch predictor prior to memset() invocation is not pre-trained for the branching sequence inside a memset() implementation.
- Memset() count values are likely to be uncorrected.

The proper measurement technique to compare memset() performance for more realistic memset() invocation scenarios will require a per-invocation technique that wraps two RDTSC around each invocation of memset().

With the per-invocation RDTSC measurement technique, the overhead of RDTSC and be pre-calibrated and post-validated outside of a measurement loop. The per-invocation technique may also consider cache warming effect by using a loop to wrap around the per-invocation measurements.

When the relevant skew factors of measurement techniques are taken into effect, the performance of memset() using REP STOSB, for count values smaller than a few hundred bytes, is generally faster than the AVX2 version for the common memset() invocation scenarios. Only in the extreme scenarios of hundreds of unrolled memset() calls, all using count values less than a few hundred bytes and with no intervening instruction stream between each pair of memset() can the AVX2 version of memset() take advantage of the training effect of the branch predictor.

15.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code

There may be situations where the data furnished by a call to memcpy/memset and subsequent instructions consuming the data can be re-arranged:

```
memcpy ( pBuf, pSrc, Cnt); // make a copy of some data with knowledge of Cnt
..... // subsequent instruction sequences are not consuming pBuf immediately
result = compute( pBuf); // memcpy result consumed here
```

When the count is known to be at least a thousand byte or more, using enhanced REP MOVSB/STOSB can provide another advantage to amortize the cost of the non-consuming code. The heuristic can be understood using a value of Cnt = 4096 and memset() as example:

- A 256-bit SIMD implementation of memset() will need to issue/execute retire 128 instances of 32-byte store operation with VMOVDQA, before the non-consuming instruction sequences can make their way to retirement.
- An instance of enhanced REP STOSB with ECX= 4096 is decoded as a long micro-op flow provided by hardware, but retires as one instruction. There are many store_data operation that must complete before the result of memset() can be consumed. Because the completion of store data operation is decoupled from program-order retirement, a substantial part of the non-consuming code stream can process through the issue/execute and retirement, essentially cost-free if the non-consuming sequence does not compete for store buffer resources.

Software that use enhanced REP MOVSB/STOSB must check its availability by verifying CPUID.(EAX=07H, ECX=0):EBX.[bit 9] reports 1.

15.16.3.5 256-bit Fetch versus Two 128-bit Fetches

On Sandy Bridge and Ivy Bridge microarchitectures, using two 16-byte aligned loads are preferred due to the 128-bit data path limitation in the memory pipeline of the microarchitecture.

To take advantage of Haswell microarchitecture's 256-bit data path microarchitecture, the use of 256-bit loads must consider the alignment implications. Instruction that fetched 256-bit data from memory should pay attention to be 32-byte aligned. If a 32-byte unaligned fetch would span across cache line boundary, it is still preferable to fetch data from two 16-byte aligned address instead.

15.16.3.6 Mixing MULX and AVX2 Instructions

Combining MULX and AVX2 instruction can further improve the performance of some common computation task, e.g. numeric conversion 64-bit integer to ascii format can benefit from the flexibility of MULX register allocation, wider YMM register, and variable packed shift primitive VPSRLVD for parallel moduli/remainder calculations.

Example 15-40[1] shows a macro sequence of AVX2 instruction to calculate one or two finite range unsigned short integer(s) into respective decimal digits, featuring VPSRLVD in conjunction with Montgomery reduction technique.

Example 15-40. Macros for Parallel Moduli/Remainder Calculation

```
static  short  quoTenThsn_mulplr_d[16] =
{  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0,  0x199a, 0, 0x28f6, 0, 0x20c5, 0, 0x1a37, 0};
static  short  mten_mulplr_d[16] = { -10, 1, -10,  1, -10,  1, -10, 1, -10, 1, -10,  1, -10,  1, -10, 1};
// macro to convert input t5 (a __m256i type) containing quotient (dword 4) and remainder
// (dword 0) into single-digit integer (between 0-9) in output y3 ( a__m256i);
//both dword element "t5" is assume to be less than 10^4, the rest of dword must be 0;
//the output is 8 single-digit integer, located in the low byte of each dword, MS digit in dword 0
#define    __ParMod10to4AVX2dw4_0( y3, t5 ) \
{ __m256i x0, x2;                    \
   x0 = _mm256_shuffle_epi32( t5, 0);  \
   x2 = _mm256_mulhi_epu16(x0, _mm256_loadu_si256( (__m256i *)  quoTenThsn_mulplr_d));\
   x2 = _mm256_srlv_epi32(  x2, _mm256_setr_epi32(0x0, 0x4, 0x7, 0xa, 0x0, 0x4, 0x7, 0xa) ); \
   (y3) = _mm256_or_si256(_mm256_slli_si256(x2, 6),   _mm256_slli_si256(t5, 2) );  \
   (y3) = _mm256_or_si256(x2, y3);\
   (y3) = _mm256_madd_epi16(y3, _mm256_loadu_si256( (__m256i *) mten_mulplr_d) ) ;\
}
// parallel conversion of dword integer (< 10^4) to 4 single digit integer in __m128i
#define    __ParMod10to4AVX2dw( x3, dw32 ) \
{ __m128i x0, x2;                    \
x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  dw32));  \
   x2 = _mm_mulhi_epu16(x0, _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d));\
   x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) ); \
   (x3) = _mm_or_si128(_mm_slli_si128(x2, 6),   _mm_slli_si128(_mm_cvtsi32_si128(  dw32), 2) );  \
      (x3) = _mm_or_si128(x2, (x3));\
   (x3) = _mm_madd_epi16((x3), _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;\
}
```

1. See Example 40 in GitHub.

Example 15-41[1] shows a helper utility and overall steps to reduce a 64-bit signed integer into a 63-bit unsigned range with reduced-range integer quotient/remainder pairs using MULX. Note that this example relies on Example 15-40 and Example 15-42.

Example 15-41. Signed 64-bit Integer Conversion Utility

```
#define    QWCG10to 80xabcc77118461cefdull

static  int   pr_cg_10to4[8] = { 0x68db8db, 0 , 0, 0, 0x68db8db, 0, 0, 0};
static  int   pr_1_m10to4[8] = { -10000, 0 , 0, 0 , 1, 0 , 0, 0};
                        (continue)
char * i64toa_avx2i(  __int64 xx, char * p)
{int cnt;
   _mm256_zeroupper();
   if( xx < 0) cnt = avx2i_q2a_u63b(-xx, p);
   else  cnt = avx2i_q2a_u63b(xx, p);
   p[cnt] = 0;
   return p;
}
   // Convert unsigned short (< 10^4) to ascii
   __inline  int ubsAvx2_Lt10k_2s_i2(int x_Lt10k, char *ps)
{int tmp;
   __m128i x0, m0, x2, x3, x4;
   if( x_Lt10k < 10) { *ps = '0' + x_Lt10k; return 1; }
   x0 = _mm_broadcastd_epi32( _mm_cvtsi32_si128(  x_Lt10k));
   // calculate quotients of divisors 10, 100, 1000, 10000
   m0 = _mm_loadu_si128( (__m128i *)  quoTenThsn_mulplr_d);
   x2 = _mm_mulhi_epu16(x0, m0);
   // u16/10, u16/100, u16/1000, u16/10000
   x2 = _mm_srlv_epi32(  x2, _mm_setr_epi32(0x0, 0x4, 0x7, 0xa) );

   // 0, u16, 0, u16/10, 0, u16/100, 0, u16/1000
   x3 = _mm_insert_epi16(_mm_slli_si128(x2, 6), (int) x_Lt10k, 1);
   x4 = _mm_or_si128(x2, x3);
   // produce 4 single digits in low byte of each dword
   x4 = _mm_madd_epi16(x4, _mm_loadu_si128( (__m128i *) mten_mulplr_d) ) ;// add bias for ascii encoding
   x2 = _mm_add_epi32( x4, _mm_set1_epi32( 0x30303030 ) );
   // pack 4 single digit into a dword, start with most significant digit
   x3 = _mm_shuffle_epi8(x2, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
   if (x_Lt10k > 999 )    {*(int *) ps = _mm_cvtsi128_si32(  x3);  return 4;}
     tmp = _mm_cvtsi128_si32(  x3);
     if (x_Lt10k > 99 ) {
        *((short *) (ps)) = (short ) (tmp >>8);
```

```
    *((short *) ps) = (short ) (tmp>>16); return 2;
    }

    }
```

1. See Example 41 in GitHub.

Example 15-42 shows the steps of numeric conversion of a 63-bit dynamic range into ascii format according to a progressive range reduction technique using a vectorized Montgomery reduction scheme. Note that this example relies on Example 15-40.

Example 15-42. Unsigned 63-bit Integer Conversion Utility

```
unsigned   avx2i_q2a_u63b (unsigned __int64 xx, char *ps)
{ __m128i  v0;
   __m256i  m0, x1, x3, x4, x5 ;
   unsigned __int64 xxi, xx2, lo64, hi64;
__int64  w;
   int j, cnt, abv16, tmp, idx, u;
       // conversion of less than 4 digits
   if ( xx < 10000 ) {
       j = ubsAvx2_Lt10k_2s_i2 ( (unsigned ) xx, ps); return j;
   } else if (xx < 100000000 ) {  // dynamic range of xx is less than 9 digits
    // conversion of 5-8 digits
   x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)xx));  // broadcast to every dword
    // calculate quotient and remainder, each with reduced range (< 10^4)
   x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
   x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
   // quotient in dw4, remainder in dw0
    m0 = _mm256_add_epi32( _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)xx), 0), x3);
   __ParMod10to4AVX2dw4_0( x3, m0); // 8 digit in low byte of each dw
    x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
    x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
          0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );

   // pack 8 single-digit integer into first 8 bytes and set rest to zeros
       x4 = _mm256_permutevar8x32_epi32( x4, _mm256_setr_epi32(0x4, 0x0, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1)  );
       tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
       BitScanForward((unsigned long *) &idx, tmp);
       cnt = 8 -idx; // actual number non-zero-leading digits to write to output
   } else  {  // conversion of 9-12 digits
       lo64 = _mulx_u64(xx, (unsigned __int64) QWCG10to8, &hi64);
       hi64 >>= 26;
       xxi = _mulx_u64(hi64, (unsigned __int64)100000000, &xx2);
       lo64 = (unsigned __int64)xx - xxi;
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
  if( hi64 < 10000) { // do digist 12-9 first
     __ParMod10to4AVX2dw(v0, (int)hi64);
     v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
      // continue conversion of low 8 digits of a less-than 12-digit value

     x5 = _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)lo64), 0);
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64));  // broadcast to every dword
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
     m0 = _mm256_add_epi32( x5, x3); // quotient in dw4, remainder in dw0
      __ParMod10to4AVX2dw4_0( x3, m0);
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
              0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
      x5 = _mm256_inserti128_si256(_mm256_setzero_si256(), _mm_shuffle_epi8(v0,
              _mm_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080)), 0);
     x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x2, 0x4, 0x0, 0x1,
              0x1, 0x1, 0x1, 0x1) );
     tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
      _BitScanForward((unsigned long *) &idx, tmp);
     cnt = 12 -idx;
  } else { // handle greater than 12 digit input value
     cnt = 0;
     if ( hi64 >  100000000) { // case of input value has more than 16 digits
        xxi =  _mulx_u64(hi64, (unsigned __int64) QWCG10to8, &xx2) ;
        abv16 = (int)(xx2 >>26);
        hi64 -= _mulx_u64((unsigned __int64) abv16, (unsigned __int64) 100000000, &xx2);
        __ParMod10to4AVX2dw(v0,  abv16);
        v0 = _mm_add_epi32( v0, _mm_set1_epi32( 0x30303030 ) );
        v0 = _mm_shuffle_epi8(v0, _mm_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080)  );
        tmp = _mm_movemask_epi8( _mm_cmpgt_epi8(v0, _mm_set1_epi32( 0x30303030 )) );
        _BitScanForward((unsigned long *) &idx, tmp);
        cnt = 4 -idx;
      }

     // conversion of lower 16 digits
     x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)hi64));  // broadcast to every dword
     x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
     x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
     m0 = _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(), _mm_cvtsi32_si128((int)hi64), 0), x3);
__ParMod10to4AVX2dw4_0( x3, m0);
     x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
```

```
     x4 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x0004080c, 0x80808080, 0x80808080, 0x80808080,
              0x0004080c, 0x80808080, 0x80808080, 0x80808080) );
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
      x1 = _mm256_broadcastd_epi32( _mm_cvtsi32_si128((int)lo64));  // broadcast to every dword
      x3 = _mm256_mul_epu32(x1, _mm256_loadu_si256( (__m256i *) pr_cg_10to4 ));
      x3 = _mm256_mullo_epi32(_mm256_srli_epi64(x3, 40), _mm256_loadu_si256( (__m256i *)pr_1_m10to4));
      m0 = _mm256_add_epi32(_mm256_inserti128_si256(_mm256_setzero_si256(),
_mm_cvtsi32_si128((int)lo64), 0), ),x3);
      __ParMod10to4AVX2dw4_0( x3, m0);
      x3 = _mm256_add_epi32( x3, _mm256_set1_epi32( 0x30303030 ) );
      x5 = _mm256_shuffle_epi8(x3, _mm256_setr_epi32(0x80808080, 0x80808080, 0x0004080c, 0x80808080,
               0x80808080, 0x80808080, 0x0004080c, 0x80808080) );
      x4 = _mm256_permutevar8x32_epi32( _mm256_or_si256(x4, x5), _mm256_setr_epi32(0x4, 0x0, 0x6, 0x2,
               0x1,  0x1, 0x1, 0x1)  );
      cnt += 16;
      if (cnt <=  16) {
         tmp = _mm256_movemask_epi8( _mm256_cmpgt_epi8(x4, _mm256_set1_epi32( 0x30303030 )) );
         _BitScanForward((unsigned long *) &idx, tmp);
         cnt -= idx;
      }
   }
}

   w = _mm_cvtsi128_si64( _mm256_castsi256_si128(x4));
   switch(cnt) {
   case 5:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
   break;
   case 6:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
   break;
   case 7:*ps = (char) (w >>8);   *(short *) (&ps[1]) = (short) (w >>16);
      *(unsigned *) (&ps[3]) = (w >>32);
   break;
   case 8: *(long long *)ps = w;
   break;
   case 9:*ps++ = (char) (w >>24);  *(long long *) (&ps[0]) =  _mm_cvtsi128_si64(
mm_srli_si128(_mm256_castsi256_si128(x4), 4));
   break;
   case 10:*(short *)ps = (short) (w >>16);
   *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
   break;
   case 11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
   *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
   break;
```

Example 15-42. Unsigned 63-bit Integer Conversion Utility (Contd.)

```
  case 12:
  break;
  case 10:*(short *)ps = (short) (w >>16);
  *(long long *) (&ps[2])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 11:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
  *(long long *) (&ps[3])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 12: *(unsigned *)ps = (unsigned int) w; *(long long *) (&ps[4])  = _mm_cvtsi128_si64(
_mm_srli_si128(_mm256_castsi256_si128(x4), 4));
  break;
  case 13:*ps++ = (char) (w >>24);  *(unsigned *) ps = (w >>32);
  *(long long *) (&ps[4]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 14:*(short *)ps = (short) (w >>16);  *(unsigned *) (&ps[2]) = (w >>32);
  *(long long *) (&ps[6]) =  _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 15:*ps = (char) (w >>8); *(short *) (&ps[1]) = (short) (w >>16);
      *(unsigned *) (&ps[3]) = (w >>32);
  *(long long *) (&ps[7])  = _mm_cvtsi128_si64( _mm_srli_si128(_mm256_castsi256_si128(x4), 8));
  break;
  case 16: _mm_storeu_si128( (__m128i *) ps, _mm256_castsi256_si128(x4));
  break;
  case 17:u = (int) _mm_cvtsi128_si64(v0);  *ps++ = (char) (u >>24);
  _mm_storeu_si128( (__m128i *) &ps[0], _mm256_castsi256_si128(x4));
  break;
  case 18:u = (int) _mm_cvtsi128_si64(v0);  *(short *)ps = (short) (u >>16);
  _mm_storeu_si128( (__m128i *) &ps[2], _mm256_castsi256_si128(x4));
  break;

  case 19:u = (int) _mm_cvtsi128_si64(v0);  *ps = (char) (u >>8); *(short *) (&ps[1]) = (short) (u >>16);
  _mm_storeu_si128( (__m128i *) &ps[3], _mm256_castsi256_si128(x4));
  break;
  case 20:u = (int) _mm_cvtsi128_si64(v0);  *(unsigned *)ps = (short) (u);
  _mm_storeu_si128( (__m128i *) &ps[4], _mm256_castsi256_si128(x4));
  break;
  }

 return cnt;
}
```

The AVX2 version of numeric conversion across the dynamic range of 3/9/17 output digits are approximately 23/57/54 cycles per input, compared to standard library implement ion's range of 85/260/560 cycles per input.

The techniques illustrated above can be extended to numeric conversion of other library, such as binary-integer-decimal (BID) encoded IEEE-754-2008 Decimal floating-point format. For BID-128 format, Example 15-42 can be adapted by adding another range-reduction stage using a pre-computed 256-bit constant to perform Montgomery reduction at modulus 10^16. The technique to construct the 256-bit constant is covered in Chapter 14, "Intel® SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing".

15.16.4 CONSIDERATIONS FOR GATHER INSTRUCTIONS

VGATHER family of instructions fetch multiple data elements specified by a vector index register containing relative offsets from a base address. Processors based on Haswell microarchitecture is the first implementation of the VGATHER instruction and a single instruction results in multiple micro-ops being executed. In the Broadwell microarchitecture, the throughput of the VGATHER family of instructions have improved significantly.

Depending on data organization and access patterns, it is possible to create equivalent code sequences without using VGATHER instruction that will execute faster and with fewer micro-ops than a single VGATHER instruction (e.g. see Section 15.5.1). Example 15-43 shows some of the situations where use of VGATHER on Haswell microarchitecture is unlikely to provide performance benefit.

Example 15-43. Access Patterns Favoring Non-VGATHER Techniques

Access Patterns	Recommended Instruction Selection
Sequential elements	Regular SIMD loads (MOVAPS/MOVUPS, MOVDQA/MOVDQU)
Fewer than 4 elements	Regular SIMD load + horizontal data-movement to re-arrange slots
Small Strides	Load all nearby elements + shuffle/permute to collected strided elements: VMOVUPD YMM0, [sequential elements] VPERMQ YMM1, YMM0, 0x08 // the even elements VPERMQ YMM2, YMM0, 0x0d // the odd elements
Transpositions	Regular SIMD loads + shuffle/permute/blend to transpose to columns

Redundant elements	Load once + shuffle/blend/logical to build data vectors in register. In this case, result[i] = x[index[i]] + x[index[i+1]], the technique below may be preferable to using multiple VGATHER:
	ymm0 <- VGATHER (x[index[k]]); // fetching 8 elements
	ymm1 <- VBLEND(VPERM (ymm0), VBROADCAST (x[indexx[k+8]]);
	ymm2 <- VPADD(ymm0, ymm1);

In other cases, using the VGATHER instruction can reduce code size and execute faster with techniques including but not limited to amortizing the latency and throughput of VGATHER, or by hoisting the fetch operations well in advance of consumer code of the destination register of those fetches. Example 15-44 lists some patterns that can benefit from using VGATHER on Haswell microarchitecture.

General tips for using VGATHER:

- Gathering more elements with a VGATHER instruction helps amortize the latency and throughput of VGATHER, and is more likely to provide performance benefit over an equivalent non-VGATHER flow. For example, the latency of 256-bit VGATHER is less than twice the equivalent 128-bit VGATHER and therefore more likely to show gains than two 128-bit equivalent ones. Also, using index size larger than data element size results in only half of the register slots utilized but not a proportional latency reduction. Therefore the dword index form of VGATHER is preferred over qword index if dwords or single-precision values are to be fetched.
- It is advantageous to hoist VGATHER well in advance of the consumer code.
- VGATHER merges the (unmasked) gathered elements with the previous value of the destination. Therefore, in cases where the previous value of the destination doesn't need to be merged (for instance, when no elements is masked off), it can be beneficial to break the dependency of the VGATHER instruction on the previous writer of the destination register (by zeroing out the register with a VXOR instruction).

Example 15-44. Access Patterns Likely to Favor VGATHER Techniques

Access Patterns	Instruction Selection

4 or more elements with unknown masks	Code with conditional element gathers typically either will not vectorize without a VGATHER instruction or provide relatively poor performance due to data-dependent mis-predicted branches.
	C code with data-dependent branches:
	if (condition[i] > 0) { result[i] = x[index[i]] }
	AVX2 equivalent sequence:
	YMM0 <- VPCMPGT (condition, zeros) // compute vector mask
	YMM2 <- VGATHER (x[YMM1], YMM0) // addr=x[YMM1], mask=YMM0

Vectorized index calculation with 8 elements	Vectorized calculations to generate the index synergizes well with the VGATHER instruction functionality.
	C code snippet:
	x[index1[i] + index2[i]]
	AVX2 equivalent:
	YMM0 <- VPADD (index1, index2) // calc vector index
	YMM1 <- VGATHER (x[YMM0], mask) // addr=x[YMM0]

Performance of the VGATHER instruction compared to a multi-instruction gather equivalent flow can vary due to:

- Differences in the base algorithm.
- Different data organization
- The effectiveness of the equivalent flow.

In performance critical applications it is advisable to evaluate both options before choosing one.

The throughput of GATHER instructions continue to improve from Broadwell to Skylake Microarchitecture. This is shown in Figure 15-11.

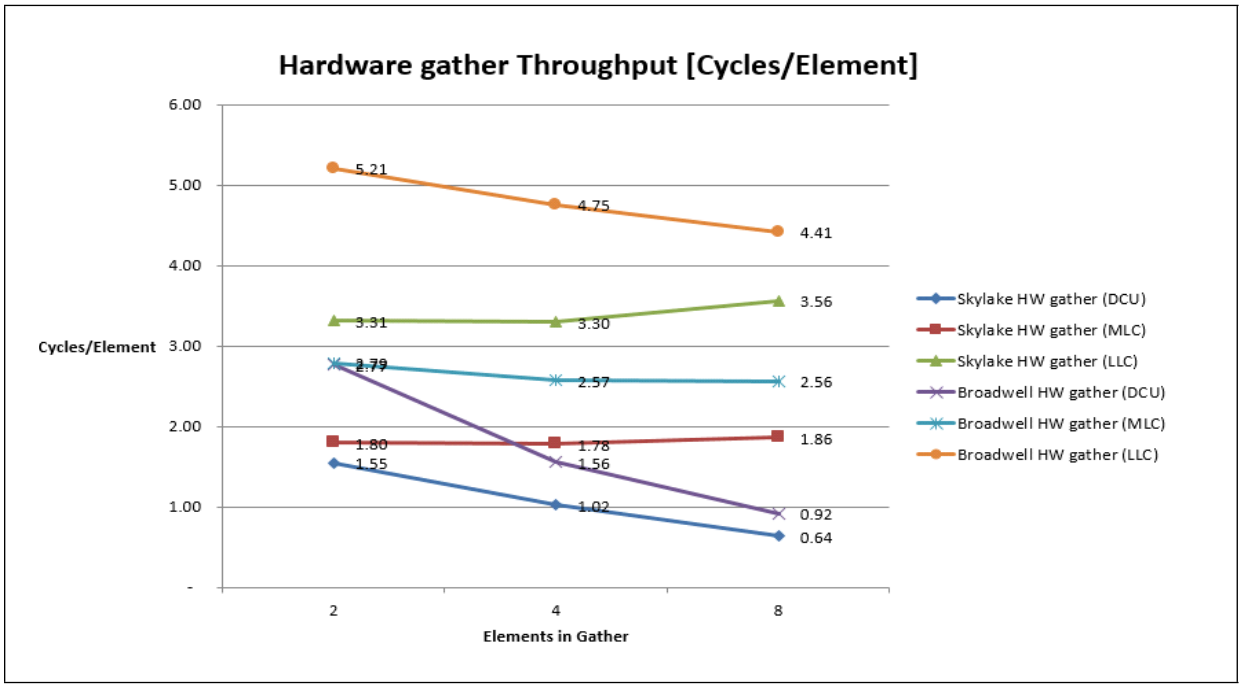


Figure 15-11. Throughput Comparison of Gather Instructions

Example 15-45 gives the asm sequence of software implementation that is equivalent to the VPGATHERD instruction. This can be used to compare the trade-off of using a hardware gather instruction or software gather sequence based on inserting an individual element.

Example 15-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD

```
mov eax, [rdi]                  // load index0
vmovd xmm0, [rsi+4*rax]             // load element0
mov eax, [rdi+4]                // load index1
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x1   // load element1
mov eax, [rdi+8]                // load index2
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x2   // load element2
mov eax, [rdi+12]               // load index3
vpinsrd xmm0, xmm0, [rsi+4*rax], 0x3   // load element3
mov eax, [rdi+16]               // load index4
vmovd xmm1, [rsi+4*rax]             // load element4
mov eax, [rdi+20]               // load index5
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x1   // load element5
mov eax, [rdi+24]               // load index6
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x2   // load element6
mov eax, [rdi+28]               // load index7
vpinsrd xmm1, xmm1, [rsi+4*rax], 0x3   // load element7
vinserti128 ymm0, ymm0, xmm1, 1       //result in ymm0
```

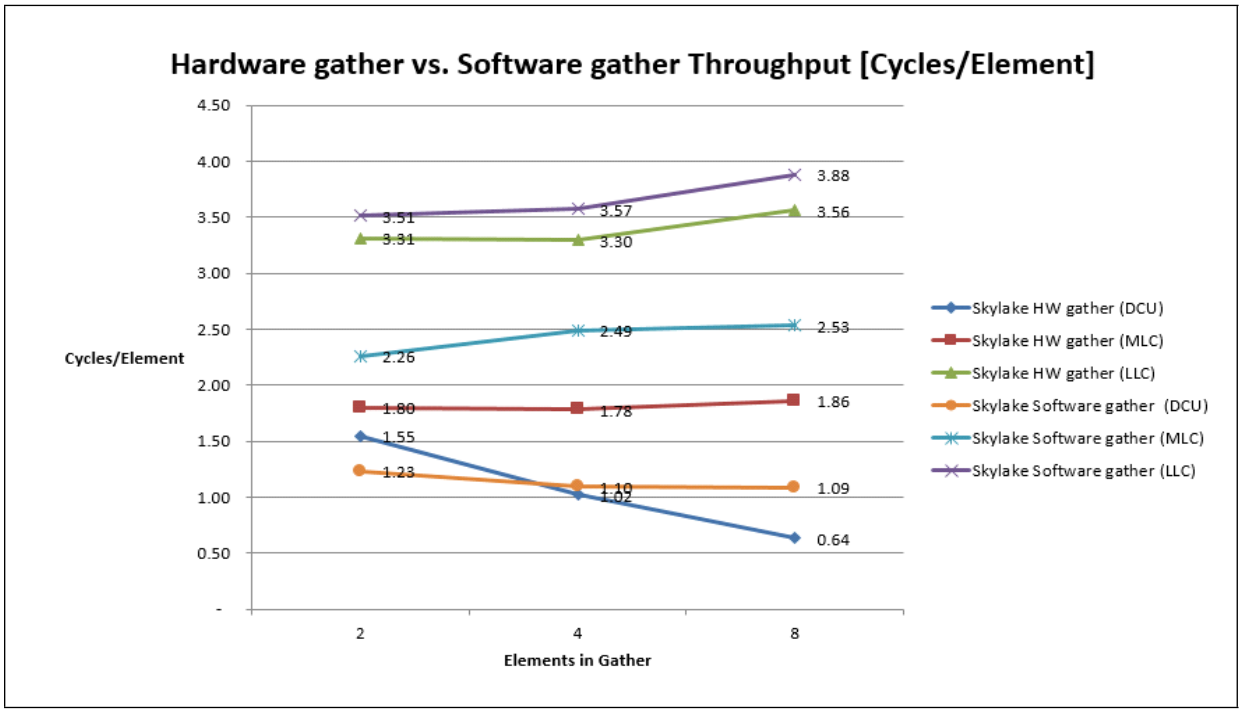


Figure 15-12. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture

Figure 15-12 compares per-element throughput using the VPGATHERD instruction versus a software gather sequence with Skylake microarchitecture as a function of cache locality of data supply. With the exception of using hardware GATHER on two data elements per instruction, the gather instruction out-performs the software sequence on Skylake microarchitecture.

If data supply locality is from memory, software sequences are likely to perform better than the hardware GATHER instruction.

15.16.4.1 Strided Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

C code:

```
    for(int i=0;i<len;i++){
        Real_buffer[i] = Complex_buffer[i].real;
        Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

Example 15-46. AOS to SOA Transformation Alternatives

1: Scalar Code	2: AVX w/ VINSRT+VSHUFPS	3: AVX2 w/ VPGATHERD

loop:
lea eax, [r10+r10*1]
movsxd rax, eax
inc r10d
mov r11d, dword ptr [rsi+rax*8]
mov dword ptr [rcx+rax*4], r11d
mov r11d, dword ptr [rsi+rax*8+0x4]
mov dword ptr [rdx+rax*4], r11d
mov r11d, dword ptr [rsi+rax*8+0x8]
mov dword ptr [rcx+rax*4+0x4], r11d
mov r11d, dword ptr [rsi+rax*8+0xc]
mov dword ptr [rdx+rax*4+0x4], r11d
cmp r10d, r8d
jl loop
	loop:
	vmovdqu xmm0, xmmword ptr [r10+rcx*8]
	vmovdqu xmm1, xmmword ptr [r10+rcx*8+0x10]
	vmovdqu xmm4, xmmword ptr [r10+rcx*8+0x40]
	vmovdqu xmm5, xmmword ptr [r10+rcx*8+0x50]
	vinserti128 ymm2, ymm0, xmmword ptr [r10+rcx*8+0x20], 0x1
	vinserti128 ymm3, ymm1, xmmword ptr [r10+rcx*8+0x30], 0x1
	vinserti128 ymm6, ymm4, xmmword ptr [r10+rcx*8+0x60], 0x1
	vinserti128 ymm7, ymm5, xmmword ptr [r10+rcx*8+0x70], 0x1
	add rcx, 0x10
	vshufps ymm0, ymm2, ymm3, 0x88
	vshufps ymm1, ymm2, ymm3, 0xdd
	vshufps ymm4, ymm6, ymm7, 0x88
	vshufps ymm5, ymm6, ymm7, 0xdd
	vmovups ymmword ptr [r9], ymm0
	vmovups ymmword ptr [r8], ymm1
	vmovups ymmword ptr [r9+0x20], ymm4
	vmovups ymmword ptr [r8+0x20], ymm5
	
	add r9, 0x40
	add r8, 0x40
	cmp rcx, rsi
	jl loop
		loop:
		lea r11, [r10+rcx*8]
		vpxor ymm5, ymm5, ymm5
		add rcx, 0x8
		vpxor ymm6, ymm6, ymm6
		vmovdqa ymm3, ymm0
		vmovdqa ymm4, ymm0
		vpgatherdd ymm5, ymmword ptr [r11+ymm2*4], ymm3
		vpgatherdd ymm6, ymmword ptr [r11+ymm1*4], ymm4
		vmovdqu ymmword ptr [r9], ymm5
		vmovdqu ymmword ptr [r8], ymm6
		add r9, 0x20
		add r8, 0x20
		cmp rcx, rsi
		jl loop

With strided access patterns, an AVX software sequence can load and shuffle on multiple elements and is the more optimal technique.

Table 15-7. Comparison of AOS to SOA with Strided Access Pattern

Microarchitecture	Scalar	VPGATHERD	AVX VINSRTF128/VSHUFFLEPS
Broadwell	1X	1.7X	4.8X
Skylake	1X	2.7X	4.9X

15.16.4.2 Adjacent Loads

This section compares using the hardware GATHER instruction versus alternative implementations of handling a variant situation of AOS to SOA transformation. In this case, AOS data are not loaded sequentially but via an index array.

C code:

```
for(int i=0;i<len;i++){
        Real_buffer[i] = Complex_buffer[Index_buffer[i]].real;
        Imaginary_buffer[i] = Complex_buffer[Index_buffer[i]].imag;
}
```

Example 15-47. Non-Strided AOS to SOA

AVX2 GATHERPD	AVX VINSRTF128 /UNPACK

loop:
vmovdqu ymm1, ymmword ptr [rsi+rdx*4]
vpaddd ymm3, ymm1, ymm1
vpaddd ymm14, ymm13, ymm3
vxorpd ymm5, ymm5, ymm5
vmovdqa ymm2, ymm0
vxorpd ymm6, ymm6, ymm6
vmovdqa ymm4, ymm0
vxorpd ymm10, ymm10, ymm10
vmovdqa ymm7, ymm0
vxorpd ymm11, ymm11, ymm11
vmovdqa ymm9, ymm0
vextracti128 xmm12, ymm14, 0x1
vextracti128 xmm8, ymm3, 0x1
vgatherdpd ymm6, ymmword ptr[r8+xmm8*8],ymm4
vgatherdpd ymm5, ymmword ptr[r8+xmm3*8],ymm2
vmovupd ymmword ptr [rcx+rdx*8], ymm5
vmovupd ymmword ptr [rcx+rdx*8+0x20], ymm6vgatherdpd ymm11, ymmword ptr[r8+xmm12*8],ymm7
vgatherdpd ymm10, ymmword ptr[r8+xmm14*8],ymm9
vmovupd ymmword ptr [rax+rdx*8], ymm10
vmovupd ymmword ptr [rax+rdx*8+0x20], ymm11
add rdx, 0x8
cmp rdx, r11
jb loop
	loop:
	movsxd r10, dword ptr [rdx+rsi*4]
	shl r10, 0x4
	movsxd r11, dword ptr [rdx+rsi*4+0x8]
	shl r11, 0x4
	vmovupd xmm0, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0x4]
	shl r10, 0x4
	vinsertf128 ymm2, ymm0, xmmword ptr [r9+r11*1], 0x1
	vmovupd xmm1, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0xc]
	shl r10, 0x4
	vinsertf128 ymm3, ymm1, xmmword ptr [r9+r10*1], 0x1
	movsxd r10, dword ptr [rdx+rsi*4+0x10]
	shl r10, 0x4
	vunpcklpd ymm4, ymm2, ymm3
	vunpckhpd ymm5, ymm2, ymm3
	vmovupd ymmword ptr [rcx], ymm4
	vmovupd xmm6, xmmword ptr [r9+r10*1]
	vmovupd ymmword ptr [rax], ymm5
	movsxd r10, dword ptr [rdx+rsi*4+0x18]
	shl r10, 0x4
	vinsertf128 ymm8, ymm6, xmmword ptr [r9+r10*1], 0x1
	movsxd r10, dword ptr [rdx+rsi*4+0x14]
	shl r10, 0x4

Example 15-47. Non-Strided AOS to SOA (Contd.)

AVX2 GATHERPD	AVX VINSRTF128 /UNPACK

	vmovupd xmm7, xmmword ptr [r9+r10*1]
	movsxd r10, dword ptr [rdx+rsi*4+0x1c]
	add rsi, 0x8
	shl r10, 0x4
	vinsertf128 ymm9, ymm7, xmmword ptr [r9+r10*1], 0x1
	vunpcklpd ymm10, ymm8, ymm9
	vunpckhpd ymm11, ymm8, ymm9
	vmovupd ymmword ptr [rcx+0x20], ymm10
	add rcx, 0x40
	vmovupd ymmword ptr [rax+0x20], ymm11
	add rax, 0x40
	cmp rsi, r8
	jl loop

With non-strided, regular access pattern of AOS to SOA, an Intel AVX software sequence that uses VINSERTF128 and interleaved packing of multiple elements can be more optimal.

Table 15-8. Comparison of Indexed AOS to SOA Transformation

Microarchitecture	VPGATHERPD	AVX VINSRTF128/VUNPCK*
Broadwell	1X	1.4X
Skylake	1.3X	1.7X

15.16.5 INTEL® AVX2 CONVERSION REMEDY TO MMX INSTRUCTION THROUGHPUT LIMITATION

In processors based on the Skylake microarchitecture, the functionality of the MMX instruction set is unchanged from prior generations. But many MMX instructions are constrained to execute to one port with half the instruction throughput relative to prior microarchitectures. The MMX instructions with throughput constraints include:

- PADDS[B/W], PADDUS[B/W], PSUBS[B/W], PSUBUS[B/W].
- PCMPGT[B/W/D], PCMPEQ[B/W/D].
- PMAX[UB/SW], PMIN[UB/SW].
- PAVG[B/W], PABS[B/W/D], PSIGN[B/W/D].

To overcome the reduction of MMX instruction throughput, conversion of asm and intrinsic code to use Intel AVX2 instruction will provide significant performance improvements. Example 15-48 shows the asm sequence using Intel AVX2 versus MMX equivalent. In the Skylake microarchitecture, the MMX code shown will execute at approximately half the speed relative to the Broadwell microarchitecture. This is due to PMAXSW/PMINSW throughput being reduced by half with the single-port restriction. When the same task is implemented with the equivalent Intel AVX2 sequence, the performance of the Intel AVX2 code on Skylake microarchitecture will be ~3.9X of the MMX code executing on the Broadwell microarchitecture.

Example 15-48. Conversion to Throughput-Reduced MMX sequence to Intel® AVX2 Alternative

MMX Code	AVX2 Code

mov rax, pIn
mov rbx, pOut
mov r8, len
mov rcx, 8
movq mm0, [rax]
movq mm1, [rax + 8]
movq mm2, mm0
movq mm3, mm1
cmp rcx, r8
jge end
loop:
movq mm4, [rax + 2*rcx]
movq mm5, [rax + 2*rcx + 8]
 	
pmaxsw mm0, mm4
pmaxsw mm1, mm5
pminsw mm2, mm4
pminsw mm3, mm5
add rcx, 8
cmp rcx, r8
jl loop
 	
end:
//Reduction
pmaxsw mm0, mm1
pshufw mm1, mm0, 0xE
pmaxsw mm0, mm1
pshufw mm1, mm0, 1
pmaxsw mm0, mm1
 	
pminsw mm2, mm3
pshufw mm3, mm2, 0xE
pminsw mm2, mm3
pshufw mm3, mm2, 1
pminsw mm2, mm3
 	
movd eax, mm0
mov WORD PTR [rbx], ax
movd eax, mm2
mov WORD PTR [rbx + 2], ax
emms
	mov rax, pIn
	mov rbx, pOut
	mov r8, len
	mov rcx, 32
	vmovdqu ymm0, ymmword ptr [rax]
	vmovdqu ymm1, ymmword ptr [rax + 32]
	vmovdqu ymm2, ymm0
	vmovdqu ymm3, ymm1
	cmp rcx, r8
	jge end
	loop:
	vmovdqu ymm4, ymmword ptr [rax + 2*rcx]
	vmovdqu ymm5, ymmword ptr [rax + 2*rcx + 32]
	vpmaxsw ymm0, ymm0, ymm4
	vpmaxsw ymm1, ymm1, ymm5
	vpminsw ymm2, ymm2, ymm4
	vpminsw ymm3, ymm3, ymm5
	add rcx, 32
	cmp rcx, r8
	jl loop
	end:
	//Reduction
	vpmaxsw ymm0, ymm0, ymm1
	vextracti128 xmm1, ymm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vpshufd xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 0xe
	vpmaxsw xmm0, xmm0, xmm1
	vpshuflw xmm1, xmm0, 1
	vpmaxsw xmm0, xmm0, xmm1
	vmovd eax, xmm0
	mov word ptr [rbx], ax
	vpminsw ymm2, ymm2, ymm3
	vextracti128 xmm1, ymm2, 1
	vpminsw xmm2, xmm2, xmm1
	vpshufd xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 0xe
	vpminsw xmm2, xmm2, xmm1
	vpshuflw xmm1, xmm2, 1
	vpminsw xmm2, xmm2, xmm1
	vmovd eax, xmm2
	mov word ptr [rbx + 2], ax

CHAPTER 16
POWER OPTIMIZATION FOR MOBILE USAGES

16.1 OVERVIEW

Mobile computing allows computers to operate anywhere, anytime. Battery life is a key factor in delivering this benefit. Mobile applications require software optimization that considers both performance and power consumption. This chapter provides background on power saving techniques in mobile processors[1] and makes recommendations that developers can leverage to provide longer battery life.

A microprocessor consumes power while actively executing instructions and doing useful work. It also consumes power in inactive states (when halted). When a processor is active, its power consumption is referred to as active power. When a processor is halted, its power consumption is referred to as static power.

ACPI 3.0 (ACPI stands for Advanced Configuration and Power Interface) provides a standard that enables intelligent power management and consumption. It does this by allowing devices to be turned on when they are needed and by allowing control of processor speed (depending on application requirements). The standard defines a number of P-states to facilitate management of active power consumption; and several C-state types[2] to facilitate management of static power consumption.

Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture implement features designed to enable the reduction of active power and static power consumption. These include:

- Enhanced Intel SpeedStep® Technology enables operating system (OS) to program a processor to transition to lower frequency and/or voltage levels while executing a workload.
- Support for various activity states (for example: Sleep states, ACPI C-states) to reduces static power consumption by turning off power to sub-systems in the processor.

Enhanced Intel SpeedStep Technology provides low-latency transitions between operating points that support P-state usages. In general, a high-numbered P-state operates at a lower frequency to reduce active power consumption. High-numbered C-state types correspond to more aggressive static power reduction. The trade-off is that transitions out of higher-numbered C-states have longer latency.

16.2 MOBILE USAGE SCENARIOS

In mobile usage models, heavy loads occur in bursts while working on battery power. Most productivity, web, and streaming workloads require modest performance investments. Enhanced Intel SpeedStep Technology provides an opportunity for an OS to implement policies that track the level of performance history and adapt the processor's frequency and voltage. If demand changes in the last 300 ms[3], the technology allows the OS to optimize the target P-state by selecting the lowest possible frequency to meet demand.

---

1. For Intel® Centrino® mobile technology and Intel® Centrino® Duo mobile technology, only processor-related techniques are covered in this manual.
2. ACPI 3.0 specification defines four C-state types, known as C0, C1, C2, C3. Microprocessors supporting the ACPI standard implement processor-specific states that map to each ACPI C-state type.
3. This chapter uses numerical values representing time constants (300 ms, 100 ms, etc.) on power management decisions as examples to illustrate the order of magnitude or relative magnitude. Actual values vary by implementation and may vary between product releases from the same vendor.

Consider, for example, an application that changes processor utilization from 100% to a lower utilization and then jumps back to 100%. The diagram in Figure 16-1 shows how the OS changes processor frequency to accommodate demand and adapt power consumption. The interaction between the OS power management policy and performance history is described below.

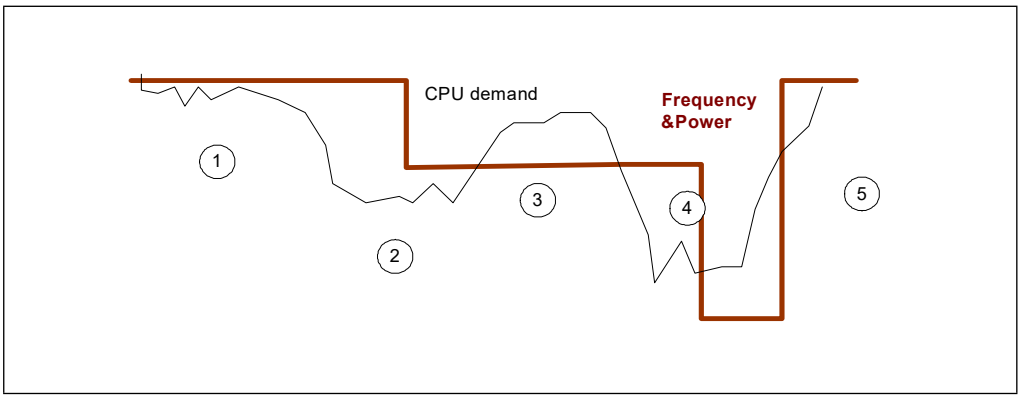


Figure 16-1. Performance History and State Transitions

1. Demand is high and the processor works at its highest possible frequency (P0).
2. Demand decreases, which the OS recognizes after some delay; the OS sets the processor to a lower frequency (P1).
3. The processor decreases frequency and processor utilization increases to the most effective level, 80-90% of the highest possible frequency. The same amount of work is performed at a lower frequency.
4. Demand decreases and the OS sets the processor to the lowest frequency, sometimes called Low Frequency Mode (LFM).
5. Demand increases and the OS restores the processor to the highest frequency.

16.2.1 INTELLIGENT ENERGY EFFICIENT SOFTWARE

With recent advances in power technology and wide range of computing scenarios demanded by end users, intelligent balance between power consumption and performance becomes more and more important. Energy efficient software plays a key role in exploring the latest hardware power savings offered by current generation architecture. Poorly-written code can prevent a system from taking advantage of new hardware features and serving the dynamic needs of end users.

A mobile platform consists of various components such as a CPU, LCD, HDD, DVD, and chipsets, which individually contribute to the power drain of the notebook. Understanding the power contribution of each major component in the platform provides a better view on the total power usage, provides guidance on optimizing power consumption, and may help software to adjust dynamic balance of power budgets between some components.

The following are a few general guidelines for energy efficient software:

- Application should leverage modern OS facility to select appropriate operating frequency instead of setting processor frequency by itself. The latter is likely to have negative impact on both power consumption and performance.
- When your application is waiting for user input or another event to happen, let your application use services that are optimized to go to idle mode quickly. The idle behavior can have a big impact on power consumption. When an application knows it will be operating in a mostly idle context, reduce the frequency of application events that wake up the processor, avoid periodic polling, and reduce the number of services that are active in memory.
- Build context awareness into applications to extend battery life further and optimal user experience.

- Architect an awareness of power consumption and/or dynamic power policy into your application for contextual usages and optimal end user experiences. Specific detail may vary across OSes. For Microsoft Windows OS consult http://www.microsoft.com/whdc/system/pnppwr/powermgmt/PMpolicy_Windows.mspx#.
- Characterize your application's power consumption. There are various techniques available to measure the power consumption of a platform:
  - Use hardware instrumentation such as Fluke NetDAQ*. This provides power measurements for each component such as CPU, HDD, and memory.
  - Use C-state residency counters. See Chapter 2, "Model-Specific Registers (MSRs)" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.
  - Study parameters such as CPU usage, kernel time, and time interrupt rate, to gain insight into the behavior of the software, which can then be related to platform power consumption if the hardware instrumentation is unavailable.

Section 16.5 provide some examples on how to relate performance with power consumption and techniques for optimizing software.

16.3 ACPI C-STATES

When computational demands are less than 100%, part of the time the processor is doing useful work and the rest of the time it is idle. For example, the processor could be waiting on an application time-out set by a Sleep() function, waiting for a web server response, or waiting for a user mouse click. Figure 16-2 illustrates the relationship between active and idle time.

When an application moves to a wait state, the OS issues a HLT instruction and the processor enters a halted state in which it waits for the next interrupt. The interrupt may be a periodic timer interrupt or an interrupt that signals an event.

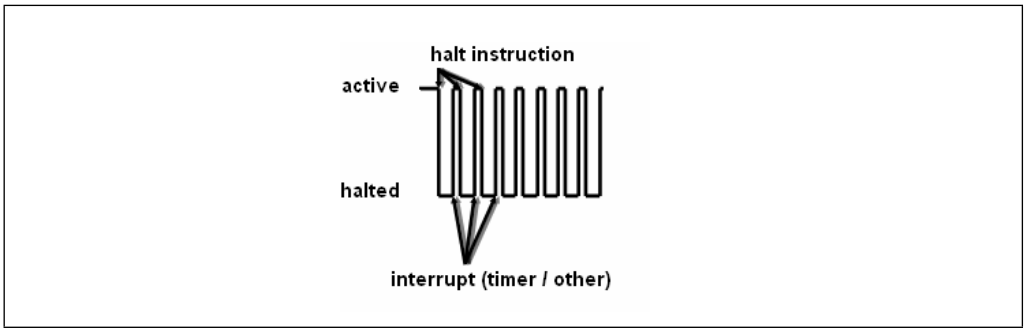


Figure 16-2. Active Time Versus Halted Time of a Processor

As shown in the illustration of Figure 16-2, the processor is in either active or idle (halted) state. ACPI defines four C-state types (C0, C1, C2 and C3). Processor-specific C states can be mapped to an ACPI C-state type via ACPI standard mechanisms. The C-state types are divided into two categories: active (C0), in which the processor consumes full power; and idle (C1-3), in which the processor is idle and may consume significantly less power.

The index of a C-state type designates the depth of sleep. Higher numbers indicate a deeper sleep state and lower power consumption. They also require more time to wake up (higher exit latency).

C-state types are described below:

- **C0:** The processor is active and performing computations and executing instructions.
- **C1**: This is the lowest-latency idle state, which has very low exit latency. In the C1 power state, the processor is able to maintain the context of the system caches.

- **C2:** This level has improved power savings over the C1 state. The main improvements are provided at the platform level.
- **C3:** This level provides greater power savings than C1 or C2. In C3, the processor stops clock generating and snooping activity. It also allows system memory to enter self-refresh mode.

The basic technique to implement OS power management policy to reduce static power consumption is by evaluating processor idle durations and initiating transitions to higher-numbered C-state types. This is similar to the technique of reducing active power consumption by evaluating processor utilization and initiating P-state transitions. The OS looks at history within a time window and then sets a target C-state type for the next time window, as illustrated in Figure 16-3:

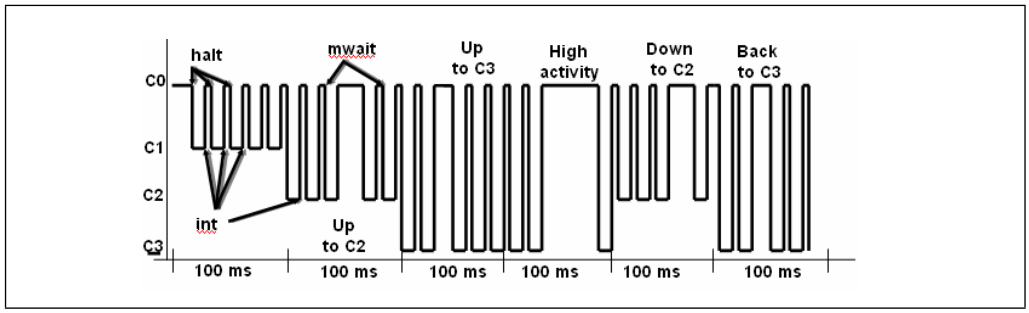


Figure 16-3. Application of C-States to Idle Time

Consider that a processor is in lowest frequency (LFM- low frequency mode) and utilization is low. During the first time slice window (Figure 16-3 shows an example that uses 100 ms time slice for C-state decisions), processor utilization is low and the OS decides to go to C2 for the next time slice. After the second time slice, processor utilization is still low and the OS decides to go into C3.

16.3.1 PROCESSOR-SPECIFIC C4 AND DEEP C4 STATES

The Pentium M, Intel Core Solo, Intel Core Duo processors, and processors based on Intel Core microarchitecture[1] provide additional processor-specific C-states (and associated sub C-states) that can be mapped to ACPI C3 state type. The processor-specific C states and sub C-states are accessible using MWAIT extensions and can be discovered using CPUID. One of the processor-specific state to reduce static power consumption is referred to as C4 state. C4 provides power savings in the following manner:

- The voltage of the processor is reduced to the lowest possible level that still allows the L2 cache to maintain its state.
- In an Intel Core Solo, Intel Core Duo processor or a processor based on Intel Core microarchitecture, after staying in C4 for an extended time, the processor may enter into a Deep C4 state to save additional static power.

The processor reduces voltage to the minimum level required to safely maintain processor context. Although exiting from a deep C4 state may require warming the cache, the performance penalty may be low enough so the benefit of longer battery life outweighs the latency of the deep C4 state.

1. Pentium M processor can be detected by CPUID signature with family 6, model 9 or 13; Intel Core Solo and Intel Core Duo processor has CPUID signature with family 6, model 14; processors based on Intel Core microarchitecture has CPUID signature with family 6, model 15.

16.3.2 PROCESSOR-SPECIFIC DEEP C-STATES AND INTEL® TURBO BOOST TECHNOLOGY

The processor-specific deep C-states are implementation dependent. Generally, the low power C-states (higher numbered C-states) have higher exit latencies. For example, when the cores are already in C7, the last level cache (L3) is flushed. The processor support auto-demotion of OS request to deep C-states (C3/C7) and demote to C1/C3 state to support flexible power-performance settings.

In addition to low-power, deep C-states, Intel Turbo Boost Technology can opportunistically boost performance in normal state (C0) by mapping p1 state to the processor's qualified high-frequency mode operation. Headroom in the system's TDP can be converted to an even higher frequency than P1 state target. When the operating system requests P0 state, the processor sets core frequencies between P1 to P0 range. A P0 state with only one core busy, achieves the maximum possible Intel Turbo Boost Technology frequency, whereas when the processor is running two to four cores the frequency is constrained by processor limitations. Under normal conditions the frequency does not go below P1, even when all cores are running.

16.3.3 PROCESSOR-SPECIFIC DEEP C-STATES FOR SANDY BRIDGE MICROARCHITECTURE

Processors based on Sandy Bridge microarchitecture implement several processor-specific C-states.

Table 16-1. ACPI C-State-Type Mappings to Processor Specific C-State of the Sandy Bridge Microarchitecture

ACPI C-State Type	Processor-Specific C-State
C0	C0
C1	C1
C2	C3
C3	C6/C7

The microarchitectural behavior of processor-specific deep C-states are implementation dependent. The following summarizes some of their key power-saving and intelligent responsive characteristics:

- For mobile platforms, while the cores are already in C7, the last level cache (L3) is flushed.
- Auto-demotion: The processor can demote OS requests to a target C-state (core C6/C7 or C3 state) to a numerically lower C-state (core C3 or C1 state) in the following cases:
  - — When history indicates that C6/C7 or C3 states are less energy efficient than C3 or C1 states.
  - — When history indicates that a deeper sleep state may impact performance.
  - — Energy inefficiency or performance degradation can occur due to the deeper C-state transition overhead occurring too frequently. Sandy Bridge microarchitecture has an enhanced algorithm that improves power gain from this feature.
- Un-demotion: An OS request to a deeper C-state can be demoted by auto-demotion, resulting in C1 or C3 states. After long residency in the demoted state, the hardware returns control back to the OS. The expectation is that in this case, the OS will repeat the deeper C-state request and hardware un-demotion will enter into the OS-requested deeper C state.

16.3.4 INTEL® TURBO BOOST TECHNOLOGY 2.0

Intel® Turbo Boost Technology 2.0 is a second generation enhancement of Intel® Turbo Boost Technology. The latter can opportunistically boost the processor core's frequency to a higher frequency above the qualified frequency depending on the TDP headroom.

The TDP of Intel Core processors based on Sandy Bridge microarchitecture include budgets for the processor core and processor graphic sub-system. Intel® Turbo Boost Technology 2.0 allows more opportunity to convert the thermal and power budget headroom to boost the processor core frequency and/or operating frequency of the processor graphic sub-system.

Energy consumption by the processor cores and/or by the processor graphic unit can be measured using a set of MSR interface[1]. Operating system requirements to support Intel Turbo Boost Technology, to use hints to optimize performance and energy bias in turbo mode operation, are described in Chapter 15, "Power and Thermal Management" of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

16.4 GUIDELINES FOR EXTENDING BATTERY LIFE

Follow the guidelines below to optimize to conserve battery life and adapt for mobile computing usage:

- Adopt a power management scheme to provide just-enough (not the highest) performance to achieve desired features or experiences.
- Avoid using spin loops.
- Reduce the amount of work the application performs while operating on a battery.
- Take advantage of hardware power conservation features using ACPI C3 state type and coordinate processor cores in the same physical processor.
- Implement transitions to and from system sleep states (S1-S4) correctly.
- Allow the processor to operate at a higher-numbered P-state (lower frequency but higher efficiency in performance-per-watt) when demand for processor performance is low.
- Allow the processor to enter higher-numbered ACPI C-state type (deeper, low-power states) when user demand for processor activity is infrequent.

16.4.1 ADJUST PERFORMANCE TO MEET QUALITY OF FEATURES

When a system is battery powered, applications can extend battery life by reducing the performance or quality of features, turning off background activities, or both. Implementing such options in an application increases the processor idle time. Processor power consumption when idle is significantly lower than when active, resulting in longer battery life.

Example of techniques to use are:

- Reducing the quality/color depth/resolution of video and audio playback.
- Turning off automatic spell check and grammar correction.
- Turning off or reducing the frequency of logging activities.
- Consolidating disk operations over time to prevent unnecessary spin-up of the hard drive.
- Reducing the amount or quality of visual animations.
- Turning off, or significantly reducing file scanning or indexing activities.

---

1. Generally, energy measurements and power management decisions based on these MSR interfaces should operate within the same processor family/model and refrain from extrapolating across different family/models or unsupported environmental conditions.

- Postponing possible activities until AC power is present.

Performance/quality/battery life trade-offs may vary during a single session, which makes implementation more complex. An application may need to implement an option page to enable the user to optimize settings for user's needs (see Figure 16-4).

To be battery-power-aware, an application may use appropriate OS APIs. For Windows XP, these include:

- **GetSystemPowerStatus:** Retrieves system power information. This status indicates whether the system is running on AC or DC (battery) power, whether the battery is currently charging, and how much battery life remains.
- **GetActivePwrScheme:** Retrieves the active power scheme (current system power scheme) index. An application can use this API to ensure that system is running best power scheme.Avoid Using Spin Loops.

Spin loops are used to wait for short intervals of time or for synchronization. The main advantage of a spin loop is immediate response time. Using the PeekMessage() in Windows API has the same advantage for immediate response (but is rarely needed in current multitasking operating systems).

However, spin loops and PeekMessage() in message loops require the constant attention of the processor, preventing it from entering lower power states. Use them sparingly and replace them with the appropriate API when possible. For example:

- When an application needs to wait for more then a few milliseconds, it should avoid using spin loops and use the Windows synchronization APIs, such as WaitForSingleObject().
- When an immediate response is not necessary, an application should avoid using PeekMessage(). Use WaitMessage() to suspend the thread until a message is in the queue.

Intel® Mobile Platform Software Development Kit provides a rich set of APIs for mobile software to manage and optimize power consumption of mobile processor and other components in the platform.

16.4.2 REDUCING AMOUNT OF WORK

When a processor is in the C0 state, the amount of energy a processor consumes from the battery is proportional to the amount of time the processor executes an active workload. The most obvious technique to conserve power is to reduce the number of cycles it takes to complete a workload (usually that equates to reducing the number of instructions that the processor needs to execute, or optimizing application performance).

Optimizing an application starts with having efficient algorithms and then improving them using Intel software development tools, such as Intel VTune Performance Analyzers, Intel compilers, and Intel Performance Libraries.

See Chapter 3, "General Optimization Guidelines" through Chapter 9, "Optimizing Cache Usage" for more information about performance optimization to reduce the time to complete application workloads.

16.4.3 PLATFORM-LEVEL OPTIMIZATIONS

Applications can save power at the platform level by using devices appropriately and redistributing the workload. The following techniques do not impact performance and may provide additional power conservation:

- Read ahead from CD/DVD data and cache it in memory or hard disk to allow the DVD drive to stop spinning.
- Switch off unused devices.
- When developing a network-intensive application, take advantage of opportunities to conserve power. For example, switch to LAN from WLAN whenever both are connected.
- Send data over WLAN in large chunks to allow the WiFi card to enter low power mode in between consecutive packets. The saving is based on the fact that after every send/receive operation, the WiFi card remains in high power mode for up to several seconds, depending on the power saving mode. (Although the purpose keeping the WiFI in high power mode is to enable a quick wake up).

- Avoid frequent disk access. Each disk access forces the device to spin up and stay in high power mode for some period after the last access. Buffer small disk reads and writes to RAM to consolidate disk operations over time. Use the GetDevicePowerState() Windows API to test disk state and delay the disk access if it is not spinning.

16.4.4 HANDLING SLEEP STATE TRANSITIONS

In some cases, transitioning to a sleep state may harm an application. For example, suppose an application is in the middle of using a file on the network when the system enters suspend mode. Upon resuming, the network connection may not be available and information could be lost.

An application may improve its behavior in such situations by becoming aware of sleep state transitions. It can do this by using the WM_POWERBROADCAST message. This message contains all the necessary information for an application to react appropriately.

Here are some examples of an application reaction to sleep mode transitions:

- Saving state/data prior to the sleep transition and restoring state/data after the wake up transition.
- Closing all open system resource handles such as files and I/O devices (this should include duplicated handles).
- Disconnecting all communication links prior to the sleep transition and re-establishing all communication links upon waking up.
- Synchronizing all remote activity, such as like writing back to remote files or to remote databases, upon waking up.
- Stopping any ongoing user activity, such as streaming video, or a file download, prior to the sleep transition and resuming the user activity after the wake up transition.

Recommendation: *Appropriately handling the suspend event enables more robust, better performing applications.*

16.4.5 USING ENHANCED INTEL SPEEDSTEP® TECHNOLOGY

Use Enhanced Intel SpeedStep Technology to adjust the processor to operate at a lower frequency and save energy. The basic idea is to divide computations into smaller pieces and use OS power management policy to effect a transition to higher P-states.

Typically, an OS uses a time constant on the order of 10s to 100s of milliseconds[1] to detect demand on processor workload. For example, consider an application that requires only 50% of processor resources to reach a required quality of service (QOS). The scheduling of tasks occurs in such a way that the processor needs to stay in P0 state (highest frequency to deliver highest performance) for 0.5 seconds and may then goes to sleep for 0.5 seconds. The demand pattern then alternates.

Thus the processor demand switches between 0 and 100% every 0.5 seconds, resulting in an average of 50% of processor resources. As a result, the frequency switches accordingly between the highest and lowest frequency. The power consumption also switches in the same manner, resulting in an average power usage represented by the equation Paverage = (Pmax+Pmin)/2.

1. The actual number may vary by OS and by OS release.

Figure 16-4 illustrates the chronological profiles of coarse-grain (> 300 ms) task scheduling and its effect on operating frequency and power consumption.

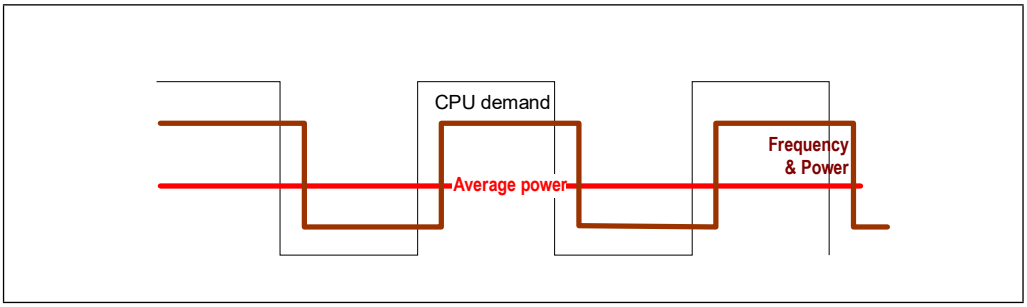


Figure 16-4. Profiles of Coarse Task Scheduling and Power Consumption

The same application can be written in such a way that work units are divided into smaller granularity, but scheduling of each work unit and Sleep() occurring at more frequent intervals (e.g. 100 ms) to deliver the same QOS (operating at full performance 50% of the time). In this scenario, the OS observes that the workload does not require full performance for each 300 ms sampling. Its power management policy may then commence to lower the processor's frequency and voltage while maintaining the level of QOS.

The relationship between active power consumption, frequency and voltage is expressed by the equation:

$$Power = \alpha * C * V^2 * F$$

In the equation: 'V' is core voltage, 'F' is operating frequency, and 'α' is the activity factor. Typically, the quality of service for 100% performance at 50% duty cycle can be met by 50% performance at 100% duty cycle. Because the slope of frequency scaling efficiency of most workloads will be less than one, reducing the core frequency to 50% can achieve more than 50% of the original performance level. At the same time, reducing the core frequency to 50% allows for a significant reduction of the core voltage.

Because executing instructions at higher P-state (lower power state) takes less energy per instruction than at P0 state, Energy savings relative to the half of the duty cycle in P0 state (Pmax /2) more than compensate for the increase of the half of the duty cycle relative to inactive power consumption (Pmin /2). The non-linear relationship between power consumption to frequency and voltage means that changing the task unit to finer granularity will deliver substantial energy savings. This optimization is possible when processor demand is low (such as with media streaming, playing a DVD, or running less resource intensive applications like a word processor, email or web browsing).

An additional positive effect of continuously operating at a lower frequency is that frequent changes in power draw (from low to high in our case) and battery current eventually harm the battery. They accelerate its deterioration.

When the lowest possible operating point (highest P-state) is reached, there is no need for dividing computations. Instead, use longer idle periods to allow the processor to enter a deeper low power mode.

16.4.6 ENABLING INTEL® ENHANCED DEEPER SLEEP

In typical mobile computing usages, the processor is idle most of the time. Conserving battery life must address reducing static power consumption.

Typical OS power management policy periodically evaluates opportunities to reduce static power consumption by moving to lower-power C-states. Generally, the longer a processor stays idle, OS power management policy directs the processor into deeper low-power C-states.

After an application reaches the lowest possible P-state, it should consolidate computations in larger chunks to enable the processor to enter deeper C-States between computations. This technique utilizes the fact that the

decision to change frequency is made based on a larger window of time than the period to decide to enter deep sleep. If the processor is to enter a processor-specific C4 state to take advantage of aggressive static power reduction features, the decision should be based on:

- Whether the QOS can be maintained in spite of the fact that the processor will be in a low-power, long-exit-latency state for a long period.
- Whether the interval in which the processor stays in C4 is long enough to amortize the longer exit latency of this low-power C state.

Eventually, if the interval is large enough, the processor will be able to enter deeper sleep and save a considerable amount of power. The following guidelines can help applications take advantage of Intel® Enhanced Deeper Sleep:

- Avoid setting higher interrupt rates. Shorter periods between interrupts may keep OSes from entering lower power states. This is because transition to/from a deep C-state consumes power, in addition to a latency penalty. In some cases, the overhead may outweigh power savings.
- Avoid polling hardware. In a ACPI C3 type state, the processor may stop snooping and each bus activity (including DMA and bus mastering) requires moving the processor to a lower-numbered C-state type. The lower-numbered state type is usually C2, but may even be C0. The situation is significantly improved in the Intel Core Solo processor (compared to previous generations of the Pentium M processors), but polling will likely prevent the processor from entering into highest-numbered, processor-specific C-state.

16.4.7 MULTICORE CONSIDERATIONS

Multicore processors deserves some special considerations when planning power savings. The dual-core architecture in Intel Core Duo processor and mobile processors based on Intel Core microarchitecture provide additional potential for power savings for multi-threaded applications.

16.4.7.1 Enhanced Intel SpeedStep® Technology

Using domain-composition, a single-threaded application can be transformed to take advantage of multicore processors. A transformation into two domain threads means that each thread will execute roughly half of the original number of instructions. Dual core architecture enables running two threads simultaneously, each thread using dedicated resources in the processor core. In an application that is targeted for the mobile usages, this instruction count reduction for each thread enables the physical processor to operate at lower frequency relative to a single-threaded version. This in turn enables the processor to operate at a lower voltage, saving battery life.

Note that the OS views each logical processor or core in a physical processor as a separate entity and computes CPU utilization independently for each logical processor or core. On demand, the OS will choose to run at the highest frequency available in a physical package. As a result, a physical processor with two cores will often work at a higher frequency than it needs to satisfy the target QOS.

For example if one thread requires 60% of single-threaded execution cycles and the other thread requires 40% of the cycles, the OS power management may direct the physical processor to run at 60% of its maximum frequency.

However, it may be possible to divide work equally between threads so that each of them require 50% of execution cycles. As a result, both cores should be able to operate at 50% of the maximum frequency (as opposed to 60%). This will allow the physical processor to work at a lower voltage, saving power.

So, while planning and tuning your application, make threads as symmetric as possible to operate at the lowest possible frequency-voltage point.

16.4.7.2 Thread Migration Considerations

Interaction of OS scheduling and multicore unaware power management policy may create some situations of performance anomaly for multi-threaded applications. The problem can arise for multithreading application that allow threads to migrate freely.

When one full-speed thread is migrated from one core to another core that has idled for a period of time, an OS without a multicore-aware P-state coordination policy may mistakenly decide that each core demands only 50% of processor resources (based on idle history). The processor frequency may be reduced by such multicore unaware P-state coordination, resulting in a performance anomaly. See Figure 16-5.

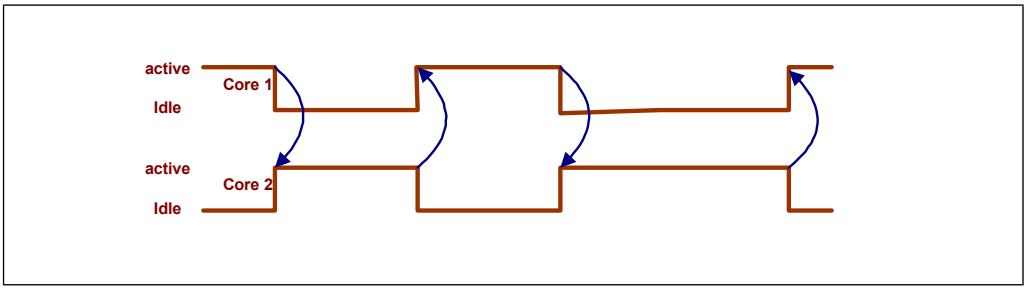


Figure 16-5. Thread Migration in a Multicore Processor

Software applications have a couple of choices to prevent this from happening:

- **Thread affinity management:** A multi-threaded application can enumerate processor topology and assign processor affinity to application threads to prevent thread migration. This can work around the issue of OS lacking multicore aware P-state coordination policy.
- **Upgrade to an OS with multicore aware P-state coordination policy:** Some newer OS releases may include multicore aware P-state coordination policy. The reader should consult with specific OS vendors.

16.4.7.3 Multicore Considerations for C-States

There are two issues that impact C-states on multicore processors.

1. Multicore-unaware C-state Coordination May Not Fully Realize Power Savings.

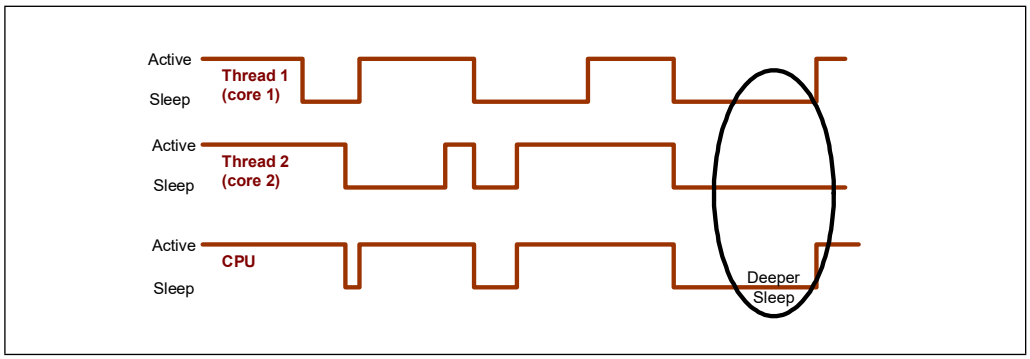


Figure 16-6. Progression to Deeper Sleep

When each core in a multicore processor meets the requirements necessary to enter a different C-state type, multicore-unaware hardware coordination causes the physical processor to enter the lowest possible C-state type (lower-numbered C state has less power saving). For example, if Core 1 meets the requirement to be in ACPI C1 and Core 2 meets requirement for ACPI C3, multicore-unaware OS coordination takes the physical processor to ACPI C1. See Figure 16-6.

2. Enabling Both Cores to Take Advantage of Intel® Enhanced Deeper Sleep.

To best utilize processor-specific C-state (e.g., Intel® Enhanced Deeper Sleep) to conserve battery life in multithreaded applications, a multi-threaded application should synchronize threads to work simultaneously and sleep simultaneously using OS synchronization primitives. By keeping the package in a fully idle state longer (satisfying ACPI C3 requirement), the physical processor can transparently take advantage of processor-specific Deep C4 state if it is available.

Multi-threaded applications need to identify and correct load-imbalances of its threaded execution before implementing coordinated thread synchronization. Identifying thread imbalance can be accomplished using performance monitoring events. Intel Core Duo processor provides an event for this purpose. The event (Serial_Execution_Cycle) increments under the following conditions:

- Core actively executing code in C0 state.
- Second core in physical processor in idle state (C1-C4).

This event enables software developers to find code that is executing serially, by comparing Serial_Execution_Cycle and Unhalted_Ref_Cycles. Changing sections of serialized code to execute into two parallel threads enables coordinated thread synchronization to achieve better power savings.

Although Serial_Execution_Cycle is available only on Intel Core Duo processors, application thread with load-imbalance situations usually remains the same for symmetric application threads and on symmetrically configured multicore processors, irrespective of differences in their underlying microarchitecture. For this reason, the technique to identify load-imbalance situations can be applied to multi-threaded applications in general, and not specific to Intel Core Duo processors.

16.5 TUNING SOFTWARE FOR INTELLIGENT POWER CONSUMPTION

This section describes some techniques for tuning software for balance of both power and performance. Most of the power optimization techniques are generic. The last sub section (Section 16.5.8) describes features specific to Sandy Bridge microarchitecture. Explore these features to optimize software for performance and corresponding power benefits.

16.5.1 REDUCTION OF ACTIVE CYCLES

Finishing the task quicker by reducing the amount of active cycles, then transfer control to the system idle loop will take advantage of modern operating system's power saving optimizations.

Reduction of active cycles can be achieved in several ways, from applying performance-oriented coding techniques discussed in Chapter 3, vectorization using SSE and/or AVX, to multi-threading.

16.5.1.1 Multithreading to Reduce Active Cycles

If given a task of some fixed amount of computational work that has thread-level parallelism, one can apply data-decomposition for multi-threading. The amount of reduction in active cycles will depend on the degree of parallelism. Similar principle can also apply to function-decomposition situations.

A balanced multi-threading implementation is more likely to achieve more optimal results in intelligent efficient performance and power saving benefits. Choosing the right synchronization primitives also has significant impact on both power and performance.

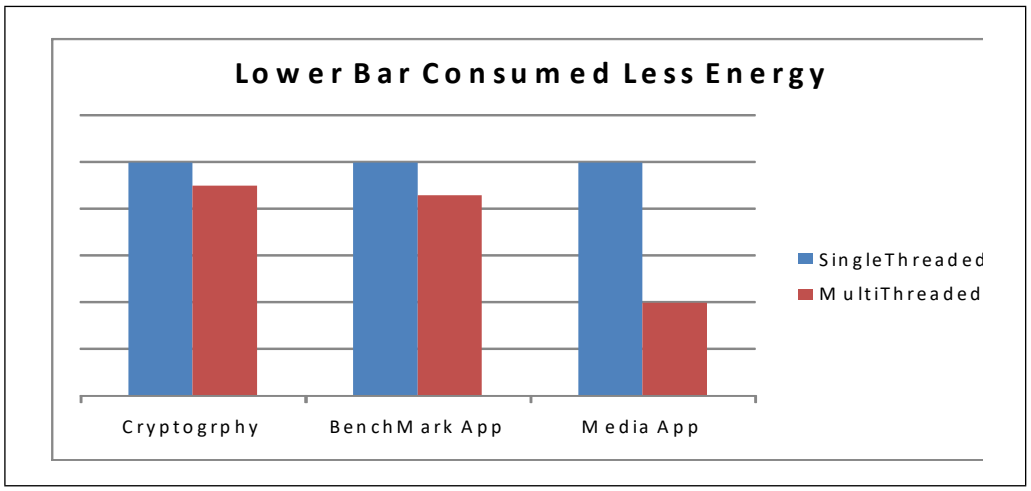


Figure 16-7. Energy Savings Due to Performance Optimization

Figure 16-7 above shows the result of a study that compares processor energy consumption of single threaded workloads with their corresponding performance-optimized implementations, using three sets of applications across different application domains. In this particular study, optimization effort in application 1 (Cryptography) achieved 2X gain in performance alone. At the same time, its energy consumption reduced about 12%. In application 3 (a media application), performance optimization efforts including multi-threading and other techniques achieved 10X performance gain. Its energy consumption reduced about 60%.

16.5.1.2 Vectorization

Use SIMD instructions can reduce the path length of completing a given computational task, often reducing active cycles. Code that performs the same operation on multiple independent data elements is a good candidate for vectorization. Vectorization techniques are typically applied to applications with loops with elements that can be processed in single instruction. Typically, the slight power increase per unit time of using SIMD instructions are compensated by much greater reduction of active cycles. The net effect is improved energy consumption.

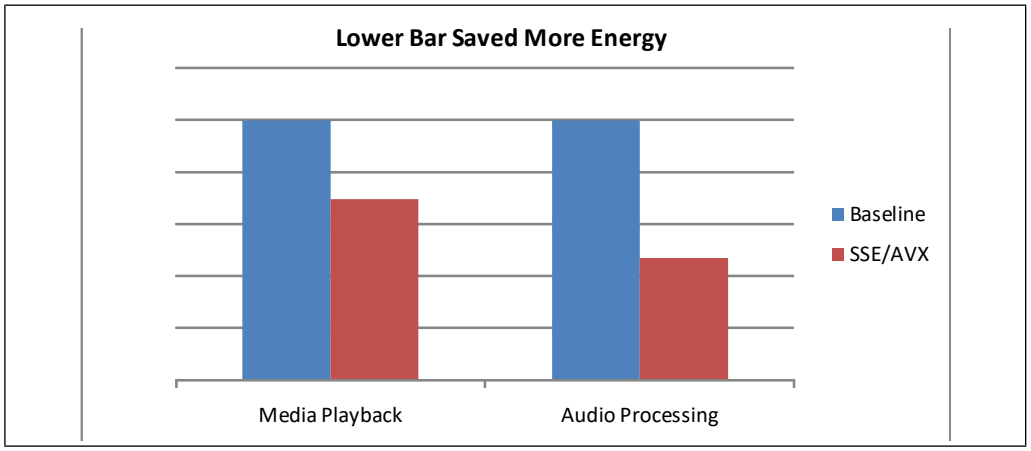


Figure 16-8. Energy Savings Due to Vectorization

Figure 16-7 shows the result of a study on the energy saving effect due to vectorization. A media playback workload achieved 2.15X speedup due to using SSE2 and SSE4 instruction sets. Another audio processing workload increased performance to ~5X by using Intel AVX instruction sets. At the same time, the latter also had better energy saving.

16.5.2 PAUSE AND SLEEP(0) LOOP OPTIMIZATION

In multi-threading implementation, a popular construct in thread synchronization and for yielding scheduling quanta to another thread waiting to carry out its task is to sit in a loop and issuing SLEEP(0).

These are typically called “sleep loops”, see Example 16-1. It should be noted that a SwitchToThread call can also be used. The “sleep loop” is common in locking algorithms and thread pools as the threads are waiting on work.

Example 16-1. Unoptimized Sleep Loop

```
while(!acquire_lock())
{ Sleep( 0 );}
do_work();
release_lock();
```

This construct of sitting in a tight loop and calling Sleep() service with a parameter of 0 is actually a polling loop with side effects:

- Each call to Sleep() experiences the expensive cost of a context switch, which can be 10000+ cycles.
- It also suffers the cost of ring 3 to ring 0 transitions, which can be 1000+ cycles.
- When there is no other thread waiting to take possession of control, this sleep loop behaves to the OS as a highly active task demanding CPU resource, preventing the OS to put the CPU into a low-power state.

Example 16-2. Power Consumption Friendly Sleep Loop Using PAUSE

```
if (!acquire_lock())
{ /* Spin on pause max_spin_count times before backing off to sleep */
  for(int j = 0; j < max_spin_count; ++j)
  {    /* intrinsic for PAUSE instruction*/
       _mm_pause();
       if (read_volatile_lock())
       {
           if (acquire_lock()) goto PROTECTED_CODE;
       }
  }
  /* Pause loop didn't work, sleep now */
  Sleep(0);
  goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
do_work();
release_lock();
```

Example 16-2 shows the technique of using PAUSE instruction to make the sleep loop power friendly.

By slowing down the “spin-wait” with the PAUSE instruction, the multi-threading software gains:

- Performance by facilitating the waiting tasks to acquire resources more easily from a busy wait.
- Power-savings by both using fewer parts of the pipeline while spinning.
- Elimination of great majority of unnecessarily executed instructions caused by the overhead of a Sleep(0) call.

In one case study, this technique achieved 4.3x of performance gain, which translated to 21% power savings at the processor and 13% power savings at platform level.

16.5.3 SPIN-WAIT LOOPS

Use the PAUSE instruction in all spin wait loops. The PAUSE instruction de-pipelines the spin-wait loop to prevent it from consuming execution resources excessively and consuming power needlessly.

When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline.

The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. However, you should try to keep spin-wait loops with PAUSE short.

16.5.4 USING EVENT DRIVEN SERVICE INSTEAD OF POLLING IN CODE

Consistently polling for devices or state changes can cause the platform to wake up and consume more power. Minimize polling whenever possible and use an event driven framework if available. If an OS provides notification services for various device state changes, such as transition from AC to battery, use them instead of polling for device state changes. Using this new event notification framework reduces the overhead for the code to poll the status of the power source, because the code can get notifications asynchronously when status changes happen.

16.5.5 REDUCING INTERRUPT RATE

High interrupt rate may have two consequences that impact processor power and performance:

- It prevents the processor package and its cores from going into deeper sleep states (C-states), which means that the system does not enable the hardware to utilize the power saving features.
- It limits the frequency to which Intel Turbo Boost Technology 2.0 can reach, and therefore the performance of other applications running on the processor degrades.

If a user session and/or an application experiences a rate of thousands of interrupts per second, it would have inhibited the processor to achieve intelligent balance between performance and saving power.

To avoid this situation minimize sporadic wakeups. Schedule all periodic activities of an application or driver into one wakeup period and reduce the interrupt rate to the minimum required.

Many media applications set a very high timer tick rate (1ms). Where possible, use the operating system default timer tick rate. If high granularity is absolutely necessary make sure the software resets the timer tick rate when the task finishes.

16.5.6 REDUCING PRIVILEGED TIME

Applications spending significant time in privileged mode lead to excessive energy use due to various reasons. Some examples are: high system call rate and IO bottlenecks. You can use Windows Perfmon to get an estimate of privileged mode time.

A high system call rate, as measured by system calls per second, indicates that the software is causing frequent kernel mode transitions. That is, the application jumps from Ring3 - user mode to Ring0 - kernel mode, frequently. A very common example of this is using an expensive synchronization call such as the Win32 API WaitForSingleObject(). This is a very important synchronization API, especially for inter-process communication. However, it enters kernel mode irrespective of whether the lock is achieved or not. For multi-threaded code with no or a short period contention on

the lock, you can use EnterCriticalSection with a spin count. The advantage of this API over WaitForSingleObject() is that it does not enter kernel mode unless there is a contention on the lock. Hence, when there is no contention, EnterCriticalSection with spin count is much cheaper to use and reduces the time spent in privilege mode.

Studies were done by taking a small test application which has four active threads on a Sandy Bridge microarchitecture-based system. The locks in the test case were implemented by using WaitForSingleObject and EnterCriticalSection. There was no contention on the lock, so each thread achieved the lock at the first attempt. As shown in the graph below, when there is no contention, using WaitForSingleObject() has negative impact on both power and performance as compared to using EnterCriticalSection().

As indicated in the following graph, using WaitForSingleObject() on an un-contended lock uses more power. Using EnterCriticalSection() provides a 10x performance gain and 60% energy reduction.

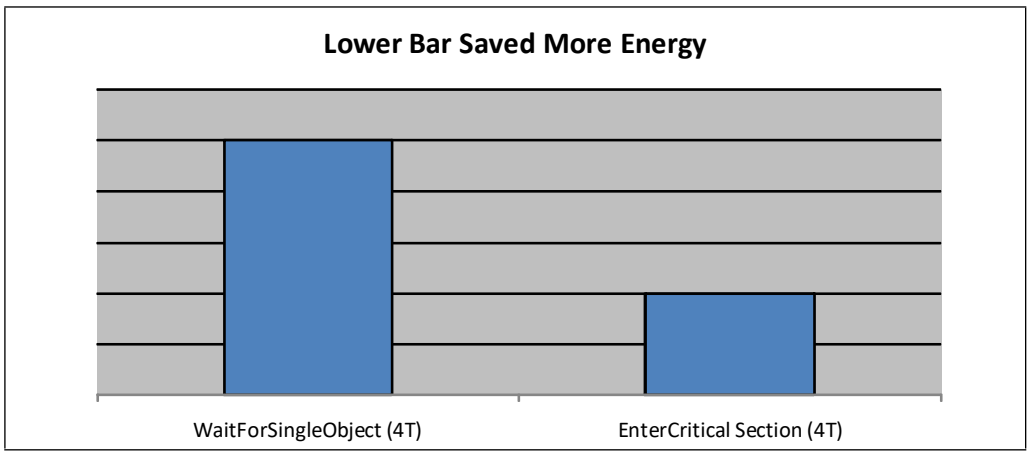


Figure 16-9. Energy-Saving Comparison of Synchronization Primitives

16.5.7 SETTING CONTEXT AWARENESS IN THE CODE

Context awareness provides a way for software to save power when energy resources are limited. The following are some examples of context awareness that you can implement to conserve power:

- When running a game on laptop on battery power, change the frame rate to 60FPS instead of running uncapped.
- Dim the display when running on battery and not in use.
- Provide easy options for the end user to turn off devices such as wireless when not connected to network

Applications can do these changes transparently when running the game on battery power, or provide hints to users how to extend battery life. For either case, the application needs to be context aware to identify when battery power is in use as opposed to AC power.

A study done with two games running at different frame rates. The blue bar represents baseline default frame rate (uncapped) for these games. The brown line represents games running at 60FPS and the yellow line represents games running at 30FPS. This study shows that capping frame rate can help reduce power consumption.

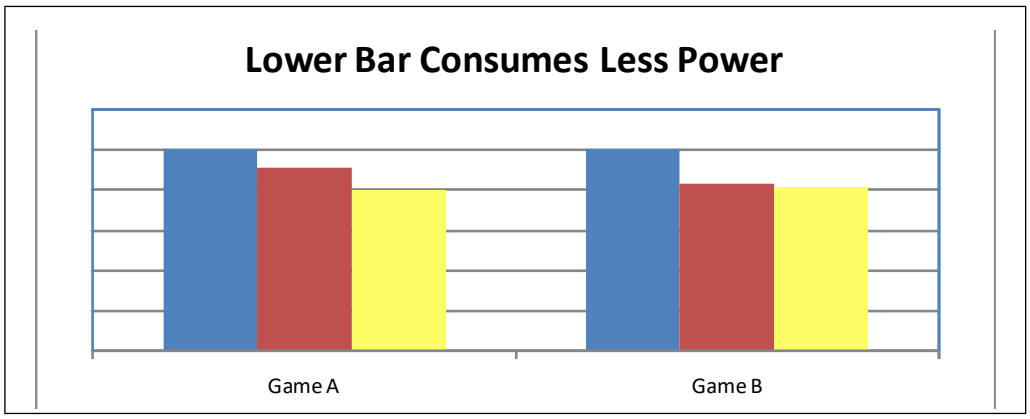


Figure 16-10. Power-Saving Comparison of Power-Source-Aware Frame Rate Configurations

16.5.8 SAVING ENERGY BY OPTIMIZING FOR PERFORMANCE

As general rule, performance optimizations that reduce the number of cycles the CPU is busy running code also save energy consumption. Here are some examples of optimizing performance for specific microarchitectural features that produced energy savings.

The additional load port feature of Sandy Bridge microarchitecture can save cycles, as demonstrated in Section 3.6. As a result the load port feature also saves power. For example, in an experiment with the kernel in Section 3.6.1.2, a sample application running on an engineering system with and without a bank conflict, the version without the bank conflict utilizes the second load port and provided a performance improvement along with 25mWHr energy savings.

Another features, the Decoded ICache and the LSD, cache micro-ops, hence eliminating power consumption by the decoders. For example, using the code alignment technique of arranging no more than three unconditional branches within an aligned 32-byte chunk (see Section 3.4.2.4) for switch statement, where we see 1.23x speedup, helps provide 1.12x power saving as well compared to the aligned version of dense unconditional branches in 32-byte chunk that can not fit the decoded ICache.

Using vectorization with Intel AVX allows processing of several elements in one cycle, hence reducing the overall processing cycles and saving energy. An example for the energy saving is shown in Figure 16-8.

16.6 PROCESSOR SPECIFIC POWER MANAGEMENT OPTIMIZATION FOR SYSTEM SOFTWARE

This section covers power management information that is processor specific. This information applies to the second generation Intel Core processors based on Sandy Bridge microarchitecture. These processors have CPUID DisplayFamily_DisplayModel signature of 06_2AH. These processor-specific capabilities may help system software to optimize/balance responsiveness and power consumption of the processor.

16.6.1 POWER MANAGEMENT RECOMMENDATION OF PROCESSOR-SPECIFIC INACTIVE STATE CONFIGURATIONS

Programming ACPI's _CST object with exit latency values appropriate to various inactive states will help OS power management to deliver optimal power reduction. Intel recommended values are model-specific.

Table 16-2 and Table 16-3 list Package C-State entry/exit latency for processors with CPUID DisplayFamily_DisplayModel signature of 06_2AH, and for two configurations of voltage regulator slew rate capabilities. Table 16-2 applies to slow VR configuration, and Table 16-3 applies to fast VR configuration. For each configuration, the VR device can operate in either a fast interrupt break mode enabled or slow interrupt break mode, depending on the setting of MSR_POWER_CTL.[bit 4]. These C-Sate entry/exit latency are not processor specifications but estimates derived from empirical measurements. There may be some situations exit latency from a core is higher than those listed in Table 16-2 and Table 16-3.

Table 16-2. C-State Total Processor Exit Latency for Client Systems with Slow VR

C-State[1]	Typical Exit Latency [2] MSR_POWER_CTL MSR.[4] =0	Worst Case Exit Latency MSR_POWER_CTL MSR.[4] =1
C1	1 µs	1 µs
C3	156 µs	80 µs
C6	181 µs	104 µs
C7	199 µs	109 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.
2. It is assumed that package is in C0 when one of the core is active.
3. Fast interrupt break mode is enabled if MSR_POWER_CTL.[4] = 1.
4. A device that connect to PCH may result in latencies equivalent to that of a slow interrupt break mode.

Table 16-3. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Fast VR

C-State[1]	Typical Worst Case Exit Latency Time (All Skus)[2] MSR_POWER_CTL MSR.[4] =0	Typical Worst Case Exit Latency Time (All Skus)[3] MSR_POWER_CTL MSR.[4] =1
C1	1 µs	1 µs
C3	156 µs	80 µs
C6	181 µs	104 µs
C7	199 µs	109 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.
2. It is assumed that package is in C0 when one of the core is active.
3. If the package is in a deeper C-states, the exit latency of Local APIC timer wake up depends on the typical core level exit latency; If the package is in C0, it may vary between typical or worst case of the respective core-level exit latency.
3. It is assumed that package is in C0 when one of the core is active.
3. If the package is in a deeper C-states, the exit latency of Local APIC timer wake up depends on the typical core level exit latency; If the package is in C0, it may vary between typical or worst case of the respective core-level exit latency.

Table 16-4 lists Core-only C-State entry/exit latency for processors with CPUID DisplayFamily_DisplayModel signature of 06_2AH, and for two configurations of voltage regulator slew rate capabilities. Core-only exit latency is not affected by MSR_POWER_CTL.[4].

Table 16-4. C-State Core-Only Exit Latency for Client Systems with Slow VR

C-State[1]	Typical Worst Case Exit Latency Time (All Skus)[2]	
C1	1 µs	1 µs
C3	21 µs	240 µs
C6	46 µs	250 µs
C7	46 µs	250 µs

NOTES:

1. These C-State Entry/Exit Latencies are Intel estimates only and not processor specifications.
2. A slow VR device refers to a device with ramp time of 10 mv/µs in fast mode and 2.5 mv/µs in slow mode.

16.6.1.1 Balancing Power Management and Responsiveness of Inactive To Active State Transitions

MSR_PKGC3_IRTL, MSR_PKGC6_IRTL, MSR_PKGC7_IRTL provide processor-specific interfaces for system software to balance power consumption and system responsiveness. System software may change budgeted values from a package inactive states to C0 during runtime to accommodate system specific requirements. For example, more aggressive timings when on battery vs. on AC power.

The exit latency is greatly impacted by the VR swing rate. Table 16-4 specifies the total interrupt response times per state (including the core component) for a “fast” exit rate (the default recommended configuration in the PCH and CPU for all events except the internal HPET and CPU timers).

Selecting a “slow” rate by the BIOS (disable the fast break event method in the POWER_CTL MSR bit 4) for various events from the PCIE will require extending the above budget respectively. Otherwise CPU may select a shallower PKG_Cstate to still meet the budget at a much slower VR swing rate.

Selecting a “slow” exit rate for various PCH-connected devices (PCH BIOS setting) will not be visible to the above latency calculation mechanism and thus result in the CPU not meeting the required latency goals.

Table 16-5. POWER_CTL MSR in Processors Based on Sandy Bridge Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1FCH	508	MSR_POWER_CTL	Core	Power Control Register.
		3:0		Reserved.
		4		FAST_Brk_Int_En. • When set to 1, enables the voltage regulator for fast slope for PCI-E interrupt wakeup events.
		63:5		Reserved.

CHAPTER 17
SOFTWARE OPTIMIZATION FOR INTEL® AVX-512 INSTRUCTIONS

As part of the family of Intel® Accelerator Engines in Intel® Xeon® Scalable processors, Intel® Advanced Vector Extensions 512 (Intel® AVX-512) provides built-in acceleration for demanding workloads that involve heavy vector-based processing. They are the following set of 512-bit instruction set extensions:

- Intel® AVX-512 Foundation (F)
  - 512-bit vector width.
  - 32 512-bit long vector registers.
  - Data expand and data compress instructions.
  - Ternary logic instruction.
  - 8 new 64-bit long mask registers.
  - Two source cross-lane permute instructions.
  - Scatter instructions.
  - Embedded broadcast/rounding.
  - Transcendental support.
- Intel® AVX-512 Conflict Detection Instructions (CD)
- Intel® AVX-512 Exponential and Reciprocal Instructions (ER)
- Intel® AVX-512 Prefetch Instructions (PF)
- Intel® AVX-512 Byte and Word Instructions (BW)
- Intel® AVX-512 Double Word and Quad Word Instructions (DQ)
  - New QWORD and Compute and Convert Instructions.
- Intel® AVX-512 Vector Length Extensions (VL)

Performance reports in this chapter are based on Data Cache Unit (DCU) resident data measurements on the Skylake Server System with Intel® Turbo-Boost technology disabled, Intel® SpeedStep® Technology disabled, core and uncore frequency set to 1.8GHz, unless otherwise specified. This fixed frequency configuration is used to isolate code change impacts from other factors.

Table 17-1. Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations

Intel® Xeon® Scalable Processors	Intel® Core™ Processors	2nd Generation Intel® Xeon® Scalable Processors	3rd Generation Intel® Xeon® Scalable Processors	4th/5th Generation Intel® Xeon® Scalable Processors
AVX/AVX2	AVX/AVX2	AVX/AVX2	AVX/AVX2	AVX/AVX2
AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ
NA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA
NA	NA	AVX512_VNNI	AVX512_VNNI	AVX512_VNNI

Table 17-1. (Contd.)Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations

Intel® Xeon® Scalable Processors	Intel® Core™ Processors	2nd Generation Intel® Xeon® Scalable Processors	3rd Generation Intel® Xeon® Scalable Processors	4th/5th Generation Intel® Xeon® Scalable Processors
NA	NA	NA	AVX512_BF16	AVX512_BF16
NA	NA	NA	NA	AVX512_VPOPCNTD, AVX512_VBM12, VAES, GFNI, VPCLMULQDQ, AVX512_BITALG
NA	NA	NA	NA	AVX512_FP16

17.1 BASIC INTEL® AVX-512 VS. INTEL® AVX2 CODING

In most cases, the main performance driver for Intel AVX-512 will be the 512-bit register width. This section demonstrates the similarity and differences between basic Intel AVX2 and Intel AVX-512 code and explains how to convert code from Intel AVX2 to Intel AVX-512 easily. The first sub section demonstrates the conversion of intrinsic code and the second sub-section of assembly code. The following sections highlight advanced aspects that require consideration and treatment when doing such conversions.

The examples in the following subsections implement a Cartesian coordinate system rotation. A point in a Cartesian coordinate system is described by the pair (x,y). The following picture demonstrates a Cartesian rotation of (x,y) by angle θ to (x',y').

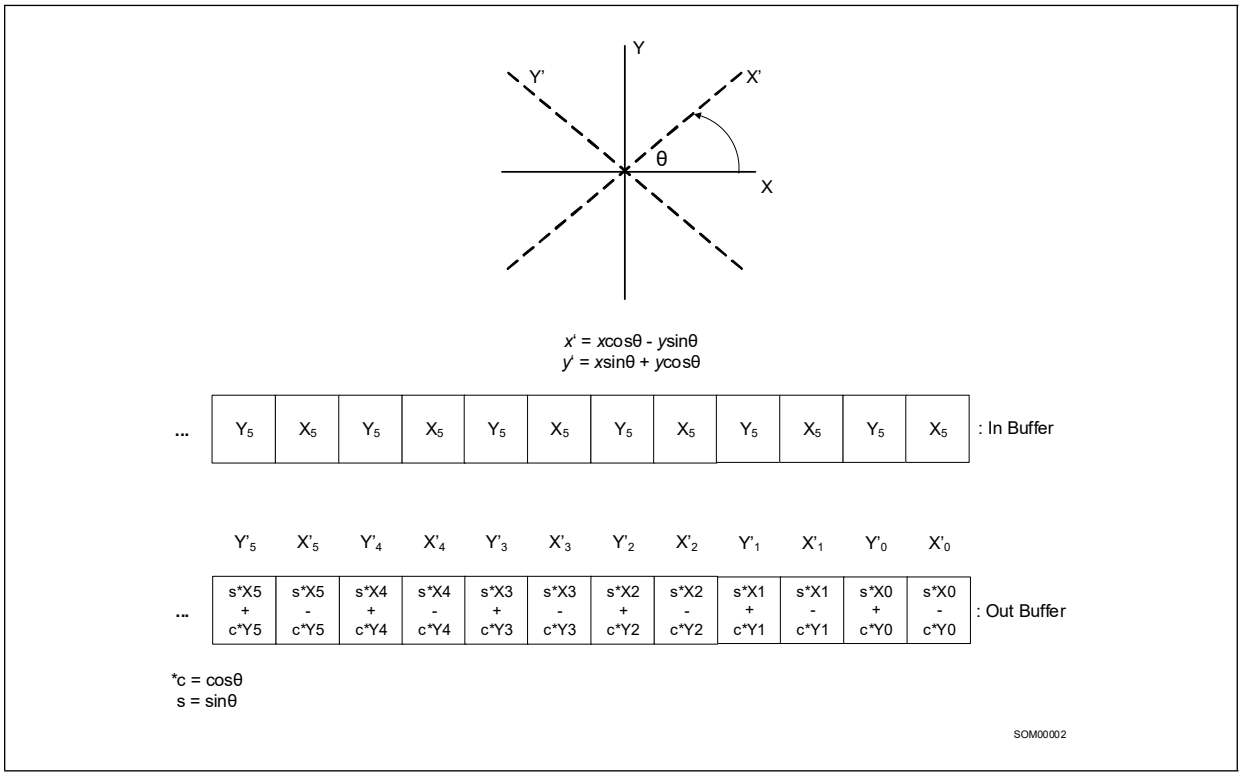


Figure 17-1. Cartesian Rotation

17.1.1 INTRINSIC CODING

The following comparison of Intel AVX2 and Intel AVX-512 shows how to convert a simple intrinsic Intel AVX2 code sequence to Intel AVX-512. This example demonstrates the Intel AVX Instruction format, 64 byte ZMM registers, dynamic and static memory allocation with data alignment of 64bytes, and the C data type representing 16 floating point elements in a ZMM register. Follow these guidelines when doing this transformation.

- Align statically and dynamically allocated buffers to 64-bytes.
- Use a double supplemental buffer size for constants.
- Change __mm256_ intrinsic name prefix with __mm512_.
- Change variable data types names from __m256 to __m512.

- Divide by 2 iteration count (double stride length).

Example 17-1. Cartesian Coordinate System Rotation with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

#include <immintrin.h>
int main()
{
int len = 3200;
//Dynamic memory allocation with 32byte
//alignment
float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);
float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);
 	
//init data
for (int i=0; i<len; i++)
pInVector[i] = 1;
 	
float cos_theta = 0.8660254037;
float sin_theta = 0.5;
 	
//Static memory allocation of 8 floats with 32byte
alignments
__declspec(align(32)) float cos_sin_theta_vec[8] =
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
sin_theta, cos_theta, sin_theta};
 	
__declspec(align(32)) float sin_cos_theta_vec[8] =
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta};
 	
//__m256 data type represents a Ymm
// register with 8 float elements
__m256 Ymm_cos_sin =
_mm256_load_ps(cos_sin_theta_vec);
	#include <immintrin.h>
	int main()
	{
	int len = 3200;
	//Dynamic memory allocation with 64byte
	//alignment
	float* pInVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	float* pOutVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	
	//init data
	for (int i=0; i<len; i++)
	pInVector[i] = 1;
	
	float cos_theta = 0.8660254037;
	float sin_theta = 0.5;
	
	//Static memory allocation of 16 floats with 64byte
	alignments
	__declspec(align(64)) float cos_sin_theta_vec[16] =
	{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta cos_theta, sin_theta,
	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta};
	
	__declspec(align(64)) float sin_cos_theta_vec[16] =
	{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
	
	//__m512 data type represents a Zmm
	// register with 16 float elements
	__m512 Zmm_cos_sin =
	_mm512_load_ps(cos_sin_theta_vec);

Example 17-1. Cartesian Coordinate System Rotation with Intrinsics (Contd.)

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

//Intel® AVX2 256bit packed single load
__m256 Ymm_sin_cos =
_mm256_load_ps(sin_cos_theta_vec);
 	
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
//processing 16 elements in an unrolled
//twice loop
for(int i=0; i<len; i+=16)
{
Ymm0 = _mm256_load_ps(pInVector+i);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector + i,Ymm3);
 	
Ymm0 = _mm256_load_ps(pInVector+i+8);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector+i+8,Ymm3);
}
 	
_mm_free(pInVector);
_mm_free(pOutVector);
 	
return 0;
}
	//Intel® AVX-512 512bit packed single load
	__m512 Zmm_sin_cos =
	_mm512_load_ps(sin_cos_theta_vec);
	__m512 Zmm0, Zmm1, Zmm2, Zmm3;
	//processing 32 elements in an unrolled
	//twice loop
	for(int i=0; i<len; i+=32)
	{
	Zmm0 = _mm512_load_ps(pInVector+i);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2,Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector + i,Zmm3);
	
	Zmm0 = _mm512_load_ps(pInVector+i+16);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector+i+16,Zmm3);
	}
	_mm_free(pInVector);
	_mm_free(pOutVector);
	
	return 0;
	}

Baseline	Speedup: 1.95x

17.1.2 ASSEMBLY CODING

Similar to the intrinsic porting guidelines, assembly porting guidelines are listed below:

- Align statically and dynamically allocated buffers to 64-bytes.
- Double the supplemental buffer sizes if needed.
- Add a “v” prefix to instruction names.
- Change register names from ymm to zmm.
- Divide the iteration count by two (or double stride length).

Example 17-2. Cartesian Coordinate System Rotation with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code

#include <immintrin.h>
int main()
{
int len = 3200;
//Dynamic memory allocation with 32byte alignment
float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);
float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);
 	
//init data
for (int i=0; i<len; i++)
pInVector[i] = 1;
 	
float cos_theta = 0.8660254037;
float sin_theta = 0.5;
 	
//Static memory allocation of 8 floats with 32byte alignments
__declspec(align(32)) float cos_sin_theta_vec[8] = {cos_theta, sin_theta,
cos_theta, sin_theta, cos_theta, sin_theta, cos_theta, sin_theta};
 	
__declspec(align(32)) float sin_cos_theta_vec[8] = {sin_theta, cos_theta, sin_theta, cos_theta, sin_theta, cos_theta, sin_theta, cos_theta};
 	
__asm
{
mov rax,pInVector
mov r8,pOutVector
// Load into a ymm register of 32 bytes
vmovups ymm3, ymmword ptr[cos_sin_theta_vec]
vmovups ymm4, ymmword ptr[sin_cos_theta_vec]
 	
mov edx, len
shl edx, 2
xor ecx, ecx
	#include <immintrin.h>
	int main()
	{
	int len = 3200;
	//Dynamic memory allocation with 64byte alignment
	float* pInVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	float* pOutVector = (float *)
	_mm_malloc(len*sizeof(float),64);
	
	//init data
	for (int i=0; i<len; i++)
	pInVector[i] = 1;
	
	float cos_theta = 0.8660254037;
	float sin_theta = 0.5;
	
	//Static memory allocation of 16 floats with 64byte alignments
	__declspec(align(64)) float cos_sin_theta_vec[16] = {cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta};
	
	__declspec(align(64)) float sin_cos_theta_vec[16] = {sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta, sin_theta cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
	__asm
	{
	mov rax,pInVector
	mov r8,pOutVector
	// Load into a zmm register of 64 bytes
	vmovups zmm3, zmmword ptr[cos_sin_theta_vec]
	vmovups zmm4, zmmword ptr[sin_cos_theta_vec]
	
	mov edx, len
	shl edx, 2
	xor ecx, ecx

Example 17-2. Cartesian Coordinate System Rotation with Assembly (Contd.)

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code

loop1:
vmovsldup ymm0, [rax+rcx]
vmovshdup ymm1, [rax+rcx]
vmulps ymm1, ymm1, ymm4
vfmaddsub213ps ymm0, ymm3, ymm1
// 32 byte store from a ymm register
vmovaps [r8+rcx], ymm0
 	
vmovsldup ymm0, [rax+rcx+32]
vmovshdup ymm1, [rax+rcx+32]
vmulps ymm1, ymm1, ymm4
vfmaddsub213ps ymm0, ymm3, ymm1
// offset 32 bytes from previous store
vmovaps [r8+rcx+32], ymm0
 	
// Processed 64bytes in this loop
// (the code is unrolled twice)
add ecx, 64
cmp ecx, edx
jl loop1
}
 	
_mm_free(pInVector);
_mm_free(pOutVector);
 	
return 0;
}
	loop1:
	vmovsldup zmm0, [rax+rcx]
	vmovshdup zmm1, [rax+rcx]
	vmulps zmm1, zmm1, zmm4
	vfmaddsub213ps zmm0, zmm3, zmm1
	// 64 byte store from a zmm register
	vmovaps [r8+rcx], zmm0
	
	vmovsldup zmm0, [rax+rcx+64]
	vmovshdup zmm1, [rax+rcx+64]
	vmulps zmm1, zmm1, zmm4
	vfmaddsub213ps zmm0, zmm3, zmm1
	// offset 64 bytes from previous store
	vmovaps [r8+rcx+64], zmm0
	
	// Processed 128bytes in this loop
	// (the code is unrolled twice)
	add ecx, 128
	cmp ecx, edx
	jl loop1
	}
	
	_mm_free(pInVector);
	_mm_free(pOutVector);
	
	return 0;
	}

Baseline	Speedup: 1.95x

17.2 MASKING

Intel AVX-512 instructions which use the Extended VEX coding scheme (EVEX) encode a predicate operand to conditionally control per-element computational operation and update the result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers, 64 bits each. From this set of 8 architectural registers, only k1 through k7 can be addressed as the predicate operand; k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

A predicate operand can be used to enable memory fault-suppression for some instructions with a memory source operand.

As a predicate operand, the opmask registers contain one bit to govern the operation / update of each data element of a vector register. Masking is supported on Skylake microarchitecture for instructions with all data sizes:

- Byte (int8)
- Word (int16)
- Single-precision floating-point (float32)
- Integer doubleword (int32)

- Double precision floating-point (float64)
- Integer quadword (int64)

Therefore, a vector register holds either 8, 16, 32 or 64 elements; accordingly, the length of a vector mask register is 64 bits. Masking on Skylake microarchitecture is also enabled for all vector length values: 128-bit, 256-bit and 512-bit. Each instruction accesses only the number of least significant mask bits needed based on its data type and vector length. For example, Intel AVX-512 instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 (i.e., 512/64) least significant bits of the opmask register.

An opmask register affects an Intel AVX-512 instruction at per-element granularity. So, any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in Intel AVX-512 has the following properties:

- The instruction's operation is only performed for an element if the corresponding opmask bit is set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value may be preserved (merging-masking) or zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a powerful construct to implement control-flow predication, since the mask provides a merging behavior for Intel AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior removes the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory, only accept merging-masking.

The per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- Generated as a result of a vector instruction (CMP, FPCLASS, etc.).
- Loaded from memory.
- Loaded from GPR register.
- Modified by mask-to-mask operations.

17.2.1 MASKING EXAMPLE

The masked instructions conditionally operate with packed data elements, depending on the mask bits associated with each data element. The mask bit for each data element is the corresponding bit in the mask register.

When performing a mask instruction, the returned value is 0 for elements which have a corresponding mask value of 0. The corresponding value in the destination register depends on the zeroing flag:

- If the flag is set, the memory location is filled with zeros.
- If the flag is not set, the values in memory location can are preserved.

The following figures show an example for a mask move from one register to another when using merging masking.

```
vmovaps  zmm1 {k1},  zmm0
```

The destination register before instruction execution is shown in Figure 17-2, 17-3 and 17-4.

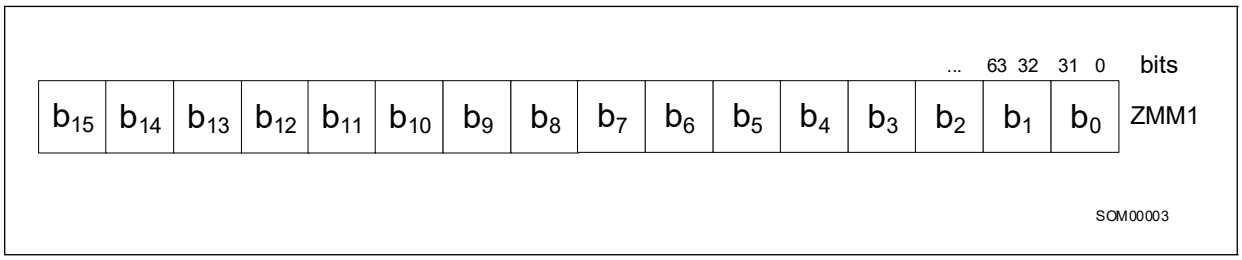


Figure 17-2. Mask Move When Using Merging Masking

The operation is as follows.

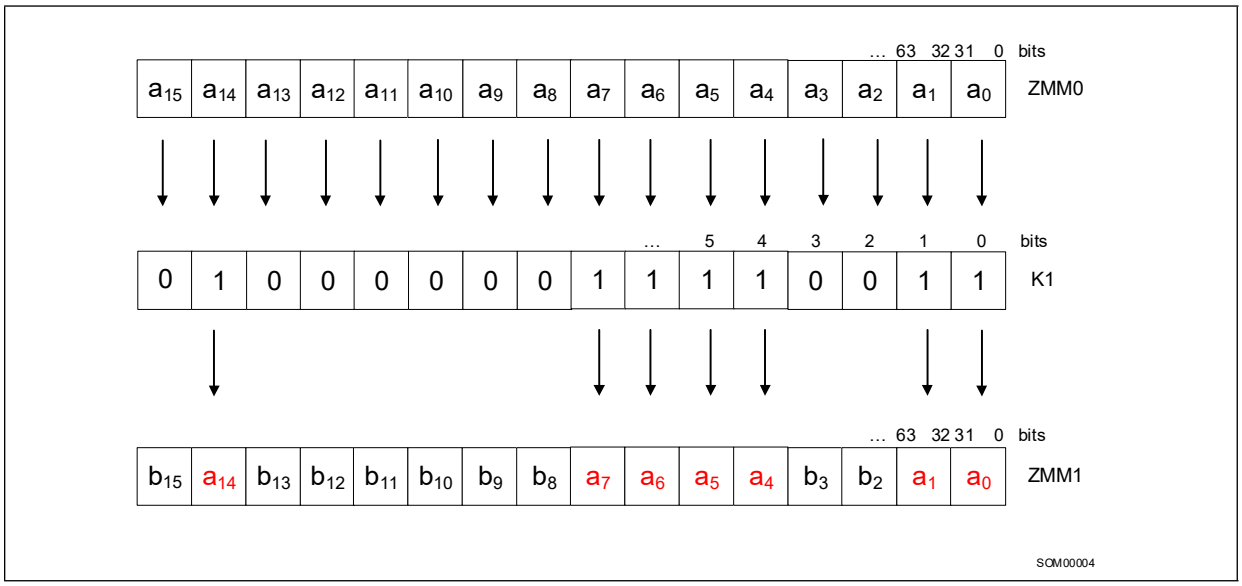


Figure 17-3. Mask Move Operation When Using Merging Masking

The result of the execution with zeroing masking is (notice the {z} in the instruction):

vmovaps zmm1 {k1}{z}, zmm0

.

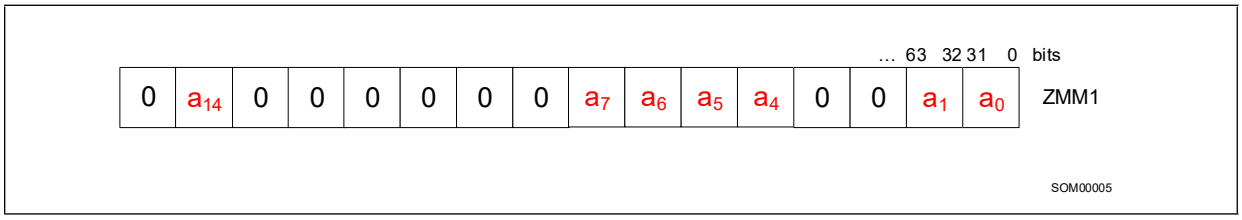


Figure 17-4. Result of Execution with Zeroing Masking

Notice that merging masking operations has a dependency on the destination, but zeroing masking is free of such dependency.

The following example shows how masking could be done with Intel AVX-512 in contrast to Intel AVX2.

C Code:

```
const int N = miBufferWidth;
const double* restrict a = A;
const double* restrict b = B;
double* restrict c = Cref;

for (int i = 0; i < N; i++){
        double res = b[i];
        if(a[i] > 1.0){
            res = res * a[i];
    }
        c[i] = res;
}
```

Example 17-3. Masking with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

for (int i = 0; i < N; i+=32){
__m256d aa, bb, mask;
#pragma unroll(8)
for (int j = 0; j < 8; j++){
aa = _mm256_loadu_pd(a+i+j*4);
bb = _mm256_loadu_pd(b+i+j*4);
mask = _mm256_cmp_pd(_mm256_set1_pd(1.0), aa, 1);
aa = _mm256_and_pd(aa, mask); // zero the false values
aa = _mm256_mul_pd(aa, bb);
bb = _mm256_blendv_pd(bb, aa, mask);
_mm256_storeu_pd(c+4*j, bb);
}
 	
c += 32;
}
	for (int i = 0; i < N; i+=32){
	__m512d aa, bb;
	__mmask8 mask;
	#pragma unroll(4)
	for (int j = 0; j < 4; j++){
	aa = _mm512_loadu_pd(a+i+j*8);
	bb = _mm512_loadu_pd(b+i+j*8);
	mask = _mm512_cmp_pd_mask(_mm512_set1_pd(1.0), aa, 1);
	bb = _mm512_mask_mul_pd(bb, mask, aa, bb);
	_mm512_storeu_pd(c+8*j, bb);
	}
	
	c += 32;
	}

Baseline	Speedup: 2.9x

Example 17-4. Masking with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code

mov rax, a
mov r11, b
mov r8, N
shr r8, 5
mov rsi, c
 	
xor rcx, rcx
xor r9, r9
 	
loop:
vmovupd ymm1, ymmword ptr [rax+rcx*8]
inc r9d
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x20]
vmovupd ymm2, ymmword ptr [r11+rcx*8]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x20]
vmovupd ymm11, ymmword ptr [rax+rcx*8+0x40]
vmovupd ymm12, ymmword ptr [r11+rcx*8+0x40]
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vcmppd ymm14, ymm0, ymm11, 0x1
vandpd ymm16, ymm1, ymm4
vandpd ymm17, ymm6, ymm9
vmulpd ymm3, ymm16, ymm2
vmulpd ymm8, ymm17, ymm7
vmovupd ymm1, ymmword ptr [rax+rcx*8+0x60]
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x80]
vblendvpd ymm5, ymm2, ymm3, ymm4
vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymm2, ymmword ptr [r11+rcx*8+0x60]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x80]
vmovupd ymmword ptr [rsi], ymm5
vmovupd ymmword ptr [rsi+0x20], ymm10
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vandpd ymm18, ymm11, ymm14
vandpd ymm19, ymm1, ymm4
vandpd ymm20, ymm6, ymm9
vmulpd ymm13, ymm18, ymm12
vmulpd ymm3, ymm19, ymm2
vmulpd ymm8, ymm20, ymm7
vmovupd ymm11, ymmword ptr [rax+rcx*8+0xa0]
vmovupd ymm1, ymmword ptr [rax+rcx*8+0xc0]
vmovupd ymm6, ymmword ptr [rax+rcx*8+0xe0]
vblendvpd ymm15, ymm12, ymm13, ymm14
vblendvpd ymm5, ymm2, ymm3, ymm4
vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymm12, ymmword ptr [r11+rcx*8+0xa0]
vmovupd ymm2, ymmword ptr [r11+rcx*8+0xc0]
vmovupd ymm7, ymmword ptr [r11+rcx*8+0xe0]
vmovupd ymmword ptr [rsi+0x40], ymm15
	mov rax, a
	mov r11, b
	mov r8, N
	shr r8, 5
	mov rsi, c
	
	xor rcx, rcx
	xor r9, r9
	mov rdi, 1
	cvtsi2sd xmm8, rdi
	vbroadcastsd zmm8, xmm8
	
	loop:
	vmovups zmm0, zmmword ptr [rax+rcx*8]
	inc r9d
	vmovups zmm2, zmmword ptr [rax+rcx*8+0x40]
	vmovups zmm4, zmmword ptr [rax+rcx*8+0x80]
	vmovups zmm6, zmmword ptr [rax+rcx*8+0xc0]
	vmovups zmm1, zmmword ptr [r11+rcx*8]
	vmovups zmm3, zmmword ptr [r11+rcx*8+0x40]
	vmovups zmm5, zmmword ptr [r11+rcx*8+0x80]
	vmovups zmm7, zmmword ptr [r11+rcx*8+0xc0]
	vcmppd k1, zmm8, zmm0, 0x1
	vcmppd k2, zmm8, zmm2, 0x1
	vcmppd k3, zmm8, zmm4, 0x1
	vcmppd k4, zmm8, zmm6, 0x1
	vmulpd zmm1{k1}, zmm0, zmm1
	vmulpd zmm3{k2}, zmm2, zmm3
	vmulpd zmm5{k3}, zmm4, zmm5
	vmulpd zmm7{k4}, zmm6, zmm7
	vmovups zmmword ptr [rsi], zmm1
	vmovups zmmword ptr [rsi+0x40], zmm3
	vmovups zmmword ptr [rsi+0x80], zmm5
	vmovups zmmword ptr [rsi+0xc0], zmm7
	add rcx, 0x20
	add rsi, 0x100
	cmp r9d, r8d
	jb loop

Example 17-4. Masking with Assembly (Contd.)

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code

vmovupd ymmword ptr [rsi+0x60], ymm5
vmovupd ymmword ptr [rsi+0x80], ymm10
vcmppd ymm14, ymm0, ymm11, 0x1
vcmppd ymm4, ymm0, ymm1, 0x1
vcmppd ymm9, ymm0, ymm6, 0x1
vandpd ymm21, ymm11, ymm14
add rcx, 0x20
vandpd ymm22, ymm1, ymm4
vandpd ymm23, ymm6, ymm9
vmulpd ymm13, ymm21, ymm12
vmulpd ymm3, ymm22, ymm2
vmulpd ymm8, ymm23, ymm7
vblendvpd ymm15, ymm12, ymm13, ymm14
vblendvpd ymm5, ymm2, ymm3, ymm4
vblendvpd ymm10, ymm7, ymm8, ymm9
vmovupd ymmword ptr [rsi+0xa0], ymm15
vmovupd ymmword ptr [rsi+0xc0], ymm5
vmovupd ymmword ptr [rsi+0xe0], ymm10
add rsi, 0x100
cmp r9d, r8d
jb loop	

Baseline	Speedup: 2.9x

17.2.2 MASKING COST

Using masking may result in lower performance than the corresponding non-masked code. This may be caused by one of the following situations:

- An additional blend operation on each load.
- Dependency on the destination when using merge masking. This dependency does not exist when using zero masking.
- More restrictive masking forwarding rules (see Forwarding and Memory Masking for more information).

The following example shows how using merge masking creates a dependency on the destination register.

Example 17-5. Masking Example

No Masking	Merge Masking	Zero Masking
mov rbx, iter loop: vmulps zmm0, zmm9, zmm8 vmulps zmm1, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}, zmm9, zmm8 vmulps zmm1{k1}, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}{z}, zmm9, zmm8 vmulps zmm1{k1}{z}, zmm9, zmm8 dec rbx jnle loop
Baseline	Slowdown: 4x	Slowdown: Equal to baseline.

With no masking, the processor executes 2 multiplies per cycle on a 2 FMA server.

With merge masking, the processor executes 2 multiplies every 4 cycles as the multiplies in iteration N depend on the output of the multiplies in iteration N-1.

Zero masking does not have a dependency on the destination register and therefore can execute 2 multiplies per cycle on a 2 FMA server.

Recommendation: Masking has a cost, so use it only when necessary. When possible, use zero masking rather than merge masking.

17.2.3 MASKING VS. BLENDING

This section discusses the advantages and disadvantages of using blending vs. masking for conditional code.

Consider the following code:

```
for ( i=0; i<SIZE; i++ )
{
                    if ( a[i] > 0 )
                                {
                      b[i] *= 2;
                                }
                           else
                                {
                      b[i] /= 2;
                                }
}
```

Example 17-6 shows two possible compilation alternatives of the code.

- Alternative 1 uses masked code and straight-forward arithmetic processing of data.
- Alternative 2 splits code to two independent unmasked flows that are processed one after another, and then a masked move (blending), just before storing to memory.

Example 17-6. Masking vs. Blending Example 1

Alternative 1	Alternative 2

mov rax, pImage
  mov rbx, pImage1
  mov rcx, pOutImage
  mov rdx, len
  vpxord zmm0, zmm0, zmm0
mainloop:
  vmovdqa32 zmm2, [rax+rdx*4-0x40]
  vmovdqa32 zmm1, [rbx+rdx*4-0x40]
  vpcmpgtd k1, zmm1, zmm0
  knotw k2, k1
	mov rax, pImage
	  mov rbx, pImage1
	  mov rcx, pOutImage
	  mov rdx, len
	  vpxord zmm0, zmm0, zmm0
	mainloop:
	  vmovdqa32 zmm2, [rax+rdx*4-0x40]
	  vmovdqa32 zmm1, [rbx+rdx*4-0x40]
	  vpcmpgtd k1, zmm1, zmm0
	  vmovdqa32 zmm3, zmm2
	  vpslld zmm2, zmm2, 1
	  vpsrld zmm3, zmm3, 1

Example 17-6. Masking vs. Blending Example 1 (Contd.)

Alternative 1	Alternative 2
(1)vpslld zmm2 {k1}, zmm2, 1 (2)vpsrld zmm2 {k2}, zmm2, 1 (3)vmovdqa32 [rcx+rdx*4-0x40], zmm2 sub rdx, 16 jne mainloop	(1)vmovdqa32 zmm3 {k1}, zmm2 (2)vmovdqa32 [rcx+rdx*4-0x40], zmm3 sub rdx, 16 jne mainloop
Baseline cycles 1x Baseline instructions 1x	Speedup: 1.23x Instructions: 1.11x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3). That means that instruction (2) has to wait for the result of the blending of instruction (1), before starting execution, and instruction (3) needs to wait for instruction (2).

In Alternative 2, there is only one such dependency because each branch of conditional code is executed in parallel on all the data, and a mask is used for blending back to one register only before writing data back to the memory.

Blending is faster, but it does not mask exceptions, which may occur on the unmasked data.

Alternative 2 executes 11% more instructions; it provides 23% speedup in overall execution. Alternative 2 uses an extra register (zmm3). This extra register usage may cause extra latency in case of register pressure (freeing register to memory and loading it afterwards).

The following code is another example of masking vs. blending.

```
for (int i = 0;i<len;i++)
{
                                        if (a[i] > b[i]){
                                          a[i] += b[i];
                                        }
}
```

Example 17-7. Masking vs. Blending Example 2

Alternative 1	Alternative 2

mov rax,a
mov rbx,b
mov rdx,size2
loop1:
vmovdqa32 zmm1,[rax +rdx*4 -0x40]
vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
(1) vpcmpgtd k1,zmm1,zmm2
(2) vmovdqa32 zmm3{k1}{z},zmm2
(3) vpaddd zmm1,zmm1,zmm3
vmovdqa32 [rax +rdx*4 -0x40],zmm1
sub rdx,16
jne loop1
	mov rax,a
	mov rbx,b
	mov rdx,size2
	loop1:
	vmovdqa32 zmm1,[rax +rdx*4 -0x40]
	vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
	(1)vpcmpgtd k1,zmm1,zmm2
	(2)vpaddd zmm1{k1},zmm1,zmm2
	vmovdqa32 [rax +rdx*4 -0x40],zmm1
	sub rdx,16
	jne loop1

Baseline cycles 1x Baseline instructions 1x	Speedup: 1.05x Instructions: 0.87x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3).

In Alternative 2, there are only 2 instructions in the dependency chain: (1) and (2).

17.2.4 NESTED CONDITIONS / MASK AGGREGATION

Intel AVX-512 contains a set of instructions for mask operation, which enable executing all bitwise logical operators on a mask register, facilitating implementation of nested and/or multiply conditions.

In the following example, logical and (&&) is executed using a *kandw* instruction.

```
for(int iX = 0; iX < iBufferWidth; iX++)
{
        if ((*pInImage)>0 && ((*pInImage)&3)==3)
{
        *pRefImage =  (*pInImage)+5;
}
        else
{
        *pRefImage = (*pInImage);
 }

        pRefImage++;
        pInImage++;
}
```

Example 17-8. Multiple Condition Execution

Scalar	Intel® AVX2	Intel® AVX-512

mov rsi, pImage
mov rdi, pOutImage
mov rbx, len
xor rax, rax
mainloop:
mov r8d, dword ptr [rsi+rax*4]
mov r9d, r8d
cmp r8d, 0
jle label1
and r9d, 0x3
cmp r9d, 3
jne label1
add r8d, 5
label1:
mov dword ptr [rdi+rax*4], r8d
add rax, 1
cmp rax, rbx
jne mainloop
	mov rsi, pImage
	mov rdi, pOutImage
	mov rbx, len
	xor rax, rax
	vpbroadcastd ymm1, [five]
	vpbroadcastd ymm7, [three]
	vpxor ymm3, ymm3, ymm3
	mainloop:
	vmovdqa ymm0, [rsi+rax*4]
	vmovaps ymm6, ymm0
	vpcmpgtd ymm5, ymm0, ymm3
	vpand ymm6, ymm6, ymm7
	vpcmpeqd ymm6, ymm6, ymm7
	vpand ymm5, ymm5, ymm6
	vpaddd ymm4, ymm0, ymm1
	vblendvps ymm4, ymm0, ymm4,
	cmp rax, rbx
	jne mainloop
		mov rsi, pImage
		mov rdi, pOutImage
		mov rbx, len
		xor rax, rax
		vpbroadcastd zmm1, [five]
		vpbroadcastd zmm5, [three]
		vpxord zmm3, zmm3, zmm3
		mainloop:
		vmovdqa32 zmm0, [rsi+rax*4]
		vpcmpgtd k1, zmm0, zmm3
		vpandd zmm6, zmm5, zmm0
		vpcmpeqd k2, zmm6, zmm5
		kandw k1, k2, k1
		vpaddd zmm0 {k1}, zmm0, zmm1
		vmovdqa32 [rdi+rax*4], zmm0
		add rax, 16
		cmp rax, rbx
		jne mainloop

Example 17-8. Multiple Condition Execution (Contd.)

Scalar	Intel® AVX2	Intel® AVX-512
Baseline 1x	Speedup: 5x	Speedup: 11x

17.2.5 MEMORY MASKING MICROARCHITECTURE IMPROVEMENTS

Masking improvements since Broadwell microarchitecture are detailed in Example 17-2

Table 17-2. Cache Comparison Between Skylake Server Microarchitecture and Broadwell Microarchitecture

Item	Broadwell Microarchitecture	Skylake Server Microarchitecture
1	• Address of a **vmaskmov** store is considered as resolved only after the mask is known. • Loads following a masked store may be blocked, depending on the memory disambiguation predictor, until the mask value is known.	• Issue is resolved. • Address of a vmaskmov store can be resolved before mask is known.
2	• **If mask is not all 1 or all 0**: loads depending upon the masked store must wait until the store data is written to the cache. • **If mask is all 1**: the data can be forwarded from the masked store to the dependent loads. • **If mask is all 0:** loads do not depend on the masked store.	• **If mask is not all 1 or all 0**: loads that depend on the masked store must wait until store data is written tocache. • **If mask is all 1**: data can be forwarded from the masked store to the dependent loads. • **If mask is all 0:** loads do not depend on the masked store.
3	• When including an illegal memory address range with masked loads (using the vmaskmov instruction): processor might take a multi-cycle "assist" to determine if any part of the illegal range has a one mask value. • Assist might occur even when mask was "all-zero" and seemed obvious that the load should not be executed.	**For Intel AVX-512 masking**: if mask is all-zeros then memory faults will be ignored and no assist will be issued.

17.2.6 PEELING AND REMAINDER MASKING

Accessing cache line aligned data gives better performance than accessing non-aligned data. In many cases, the address is not known in compile time, or known and not-aligned. In these cases a peeling algorithm may be proposed, to process first elements in masked mode, up to first aligned address, and then process unmasked body and masked remainder. This method increases code size, but improves data processing overall.

The following code is an example of peeling and remainder masking.

```
for (size_t i = 0; i < len; i++)
                                    pOutImage[i] = (pInImage[i] * alfa) + add_value;
```

The table below shows the difference in implementation and execution speed of two versions of the code, both working on unaligned output data array.

Example 17-9. Peeling and Remainder Masking

No peeling, unmasked body, masked remainder	Peeling, unmasked body, masked remainder

mov rbx, pOutImage // Output
mov rax, pImage // Input
mov rcx, len
mov edx, addValue
vpbroadcastd zmm0, edx
mov edx, alfa
vpbroadcastd zmm3, edx
mov rdx, rcx
sar rdx, 4 // 16 elements per iteration, RDX - number of full iterations
jz remainder // no full iterations
xor r8, r8
vmovups zmm10, [indices]
 	
mainloop:
vmovups zmm1, [rax + r8]
vfmadd213ps zmm1, zmm3, zmm0
vmovups [rbx + r8], zmm1
add r8, 0x40
sub rdx, 1
jne mainloop
 	
remainder:
// produce mask for remainder
and rcx, 0xF // number of elements in remainder
jz end // no elements in remainder
vpbroadcastd zmm2, ecx
vpcmpd k2, zmm10, zmm2, 1 //compare lower
 	
vmovups zmm1 {k2}{z}, [rax + r8]
vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
vmovups [rbx + r8] {k2}, zmm1
end:
	mov rax, pImage // Input
	mov rbx, pOutImage // Output
	mov rcx, len
	movss xmm0, addValue
	vpbroadcastd zmm0, xmm0
	movss xmm1, alfa
	vpbroadcastd zmm3, xmm1
	xor r8, r8
	xor r9, r9
	vmovups zmm10, [indices]
	vpbroadcastd zmm12, ecx
	
	peeling:
	mov rdx, rbx
	and rdx, 0x3F
	jz endofpeeling //nothing to peel
	neg rdx
	add rdx, 64 // 64 - X
	// now rdx contains the number of bytes to the closest alignment
	mov r9, rdx
	sar r9, 2 // now r9 contains number of elements in peeling
	
	vpbroadcastd zmm12, r9d
	vpcmpd k2, zmm10, zmm12, 1 //compare lower to produce mask for peeling
	
	vmovups zmm1 {k2}{z}, [rax]
	vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
	vmovups [rbx] {k2}, zmm1 //unaligned store
	
	endofpeeling:
	sub rcx, r9
	mov r8, rcx
	sar r8, 4 //number of full iterations
	jz remainder //no full iterations

Example 17-9. Peeling and Remainder Masking (Contd.)

No peeling, unmasked body, masked remainder	Peeling, unmasked body, masked remainder

	mainloop:
	  vmovups zmm1, [rax + rdx]
	  vfmadd213ps zmm1, zmm3, zmm0
	  vmovaps [rbx + rdx], zmm1 // aligned store is safe here
	!!
	  add rdx, 0x40
	  sub r8, 1
	  jne mainloop
	remainder:
	  // produce mask for remainder
	  and rcx, 0xF // number of elements in remainder
	  jz end // no elements in remainder
	  vpbroadcastd zmm2, ecx
	  vpcmpd k2, zmm10, zmm2, 1 //compare lower
	  vmovups zmm1 {k2}{z}, [rax + rdx]
	  vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
	  vmovaps [rbx + rdx] {k2}, zmm1 //aligned
	end:

Baseline 1x	Speedup: 1.04x

17.3 FORWARDING AND UNMASKED OPERATIONS

When using an unmasked store instruction, and load instruction after it, data forwarding depends on load type, size and address offset from store address, and does not depend on the store address itself (i.e., the store address does not have to be aligned to or fit into cache line, forwarding will occur for non-aligned and even line-split stores).

The figure below describes all possible cases when data forwarding will occur.

General Purpose Registers (GPR)

Load size	Offset from store address (in bytes)																														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32..63
1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N
4	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

SOM00006

Figure 17-5. Data Forwarding Cases

There are two important points to be considered when using data forwarding.

1. Data forwarding to GPR is possible only from the lower 256 bits of store instruction. Note this when loading GPR with data that has recently been written.
2. Do not use masks, as forwarding is supported only for certain masks.

17.4 FORWARDING AND MEMORY MASKING

When using masked store and load, consider the following:

- When the mask is not all-ones or all-zeroes, the load operation, following the masked store operation from the same address is blocked, until the data is written to the cache.
- Unlike GPR forwarding rules, vector loads whether or not they are masked, do not forward unless load and store addresses are exactly the same.
  - — st_mask = 10101010, ld_mask = 01010101, can forward: no, should block: yes
  - — st_mask = 00001111, ld_mask = 00000011, can forward: no, should block: yes
- When the mask is all-ones, blocking does not occur, because the data may be forwarded to the load operation.
  - — st_mask = 11111111, ld_mask = don't care, can forward: yes, should block: no
- When mask is all-zeroes, blocking does not occur, though neither does forwarding.

— st_mask = 00000000, ld_mask = don't care, can forward: no, should block: no

In summary, a masked store should be used carefully, for example, if the remainder size is known at compile time to be 1, and there is a load operation from the same cache line after it (or there is an overlap in addresses + vector lengths), it may be better to use scalar remainder processing, rather than a masked remainder block.

17.5 DATA COMPRESS

The data compress operation reads elements from an input buffer on indices specified by mask register 1's bits. The elements which have been read, are then written to the destination buffer. If the number of elements is less than the destination register size, the rest of the space is filled with zeros.

The following figure describes the data compress operation.

```
if (k[i] == 1)
{
        dest[a] = src[i];
                a++;
}
```

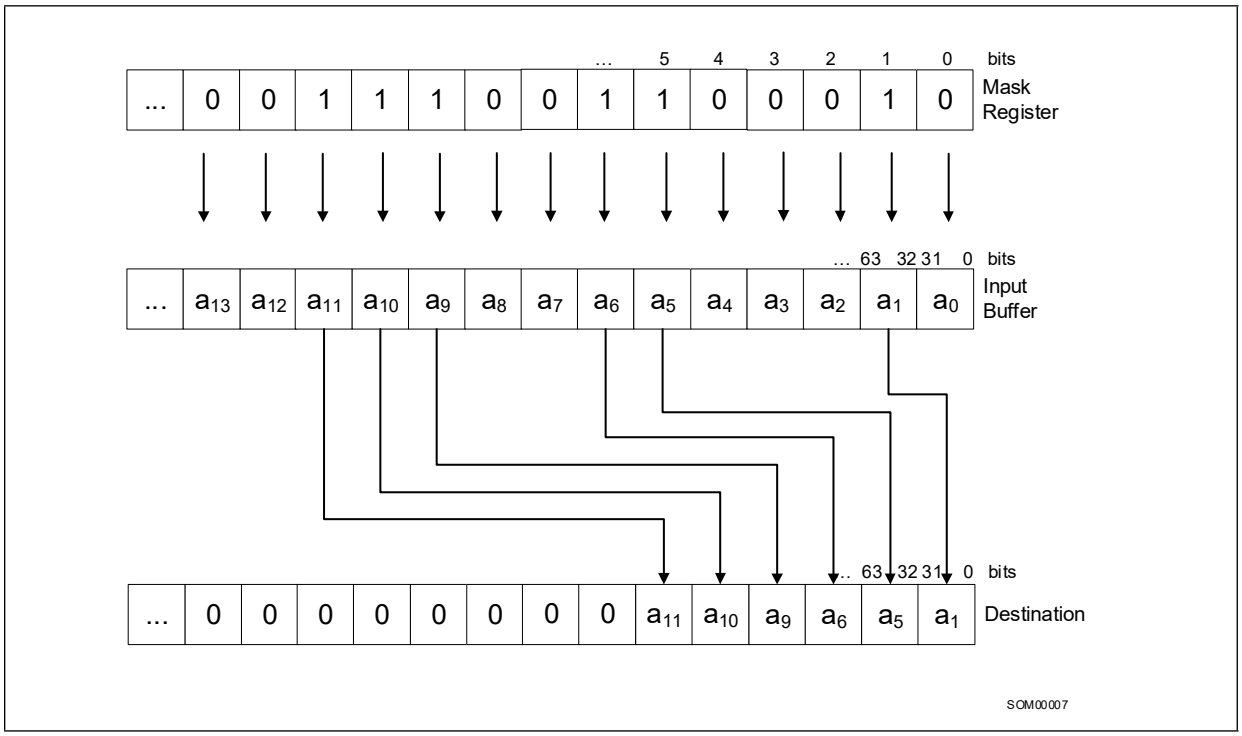


Figure 17-6. Data Compress Operation

17.5.1 DATA COMPRESS EXAMPLE

The following snippet shows collection of all positive elements from one array to another array.

```
for (int i=0; i<SIZE; i++)
{
        if ( a[i] > 0 )
        b[j++] = a[i];
}
```

Following are four implementations for the compress operation from an array of dword elements.

- **Alternative 1:** uses scalar data access and checks each element separately. If it is greater than 0 it is written to the destination array.
- **Alternative 2:** Intel AVX code that uses a shuffle instruction together with the pre-allocated and pre-initialized table with shuffle keys. The compare instruction provides the entry point number to the shuffle-key table. Then the key is loaded and the original array is shuffled according to the keys. Four elements are processed in each iteration.
- **Alternative 3:** uses the same algorithm as in Alternative 2, but uses Intel AVX2 256-bit registers, and a permutation on the dword instruction instead of using byte shuffle. Eight elements are processed in each iteration.
- **Alternative 4:** an Intel AVX-512 algorithm, which uses the *vpcompress* instruction together with the mask register as a compress key. 16 elements are processed in each iteration

Example 17-10. Comparing Intel® AVX-512 Data Compress with Alternative 1

Alternative 1: Scalar

mov rsi, source
mov rdi, dest
mov r9, len
 	
xor r8, r8
xor r10, r10
mainloop:
mov r11d, dword ptr [rsi+r8*4]
test r11d, r11d
jle m1
mov dword ptr [rdi+r10*4], r11d
inc r10
m1:
inc r8
cmp r8, r9
jne mainloop

Baseline 1x
Speedup: 2.87x

Example 17-11. Comparing Intel® AVX-512 Data Compress with Alternative 2

Alternative 2: Intel® AVX

```
        mov rsi, source
        mov rdi, dest
        mov r14, shuffle_LUT
        mov r15, write_mask
        mov r9, len

        xor r8, r8
        xor r11, r11
        vpxor xmm0, xmm0, xmm0
mainloop:
        vmovdqa xmm1, [rsi+r8*4]
        vpcmpgtd xmm2, xmm1, xmm0
        mov r10, 4
        vmovmskps r13, xmm2
        shl r13, 4
        vmovdqu xmm3, [r14+r13]
        vpshufb xmm2, xmm1, xmm3
        popcnt r13, r13
        sub r10, r13
        vmovdqu xmm3, [r15+r10*4]
        vmaskmovps [rdi+r11*4], xmm3, xmm2
        add r11, r13
        add r8, 4
        cmp r8, r9
        jne mainloop

shuffle_LUT:
   .int 0x80808080, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x80808080, 0x80808080, 0x80808080
   .int 0x07060504, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x80808080, 0x80808080
   .int 0x0b0A0908, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x80808080, 0x80808080
   .int 0x07060504, 0x0b0A0908, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x80808080
   .int 0x0F0E0D0C, 0x80808080, 0x80808080, 0x80808080
   .int 0x03020100, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x07060504, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x07060504, 0x0F0E0D0C, 0x80808080
   .int 0x0b0A0908, 0x0F0E0D0C, 0x80808080, 0x80808080
   .int 0x03020100, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x07060504, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
   .int 0x03020100, 0x07060504, 0x0b0A0908, 0x0F0E0D0C

write_mask:
   .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
   .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Speedup: 2.87x

Example 17-12. Comparing Intel® AVX-512 Data Compress with Alternative 3

Alternative 3: Intel® AVX2

```
        mov rsi, source
        mov rdi, dest
        mov r14, shuffle_LUT
        mov r15, write_mask
        mov r9, len

        xor r8, r8
        xor r11, r11
        vpxor ymm0, ymm0, ymm0
mainloop:
        vmovdqa ymm1, [rsi+r8*4]
        vpcmpgtd ymm2, ymm1, ymm0
        mov r10, 8
        vmovmskps r13, ymm2
        shl r13, 5
        vmovdqu ymm3, [r14+r13]
        vpermd ymm2, ymm3, ymm1
        popcnt r13, r13
        sub r10, r13
        vmovdqu ymm3, [r15+r10*4]
        vmaskmovps [rdi+r11*4], ymm3, ymm2
        add r11, r13
        add r8, 8
        cmp r8, r9
        jne mainloop

// The lookup table is too large to reproduce in the document. It consists of 256 rows of 8 32 bit integers.
//The first 8 and the last 8 rows are shown below.

shuffle_LUT:
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0
// Skipping 240 lines
.int 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0, 0x0
.int 0x0, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7
write_mask:
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
```

Example 17-12. (Contd.)Comparing Intel® AVX-512 Data Compress with Alternative 3 (Contd.)

Alternative 3: Intel® AVX2
.int 0x00000000, 0x00000000, 0x00000000, 0x00000000 .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
Speedup: 5.27x

Example 17-13. Comparing Intel® AVX-512 Data Compress with Alternative 4

Alternative 4: Intel® AVX-512

mov rsi, source
mov rdi, dest
mov r9, len
 	
xor r8, r8
xor r10, r10
vpxord zmm0, zmm0, zmm0
mainloop:
vmovdqa32 zmm1, [rsi+r8*4]
vpcmpgtd k1, zmm1, zmm0
vpcompressd zmm2 {k1}, zmm1
vmovdqu32 [rdi+r10*4], zmm2
kmovd r11d, k1
popcnt r12, r11
add r8, 16
add r10, r12
cmp r8, r9
jne mainloop

Speedup: 11.9x

17.6 DATA EXPAND

Data expand operations read elements from the source array (register) and put them in the destination register in the places indicated by enabled bits in the mask register. If the number of enabled bits is less than destination register size, the extra values are ignored.

```
if (k[i] == 1)
{
        dest[i] = src[a];
        a++;
}
```

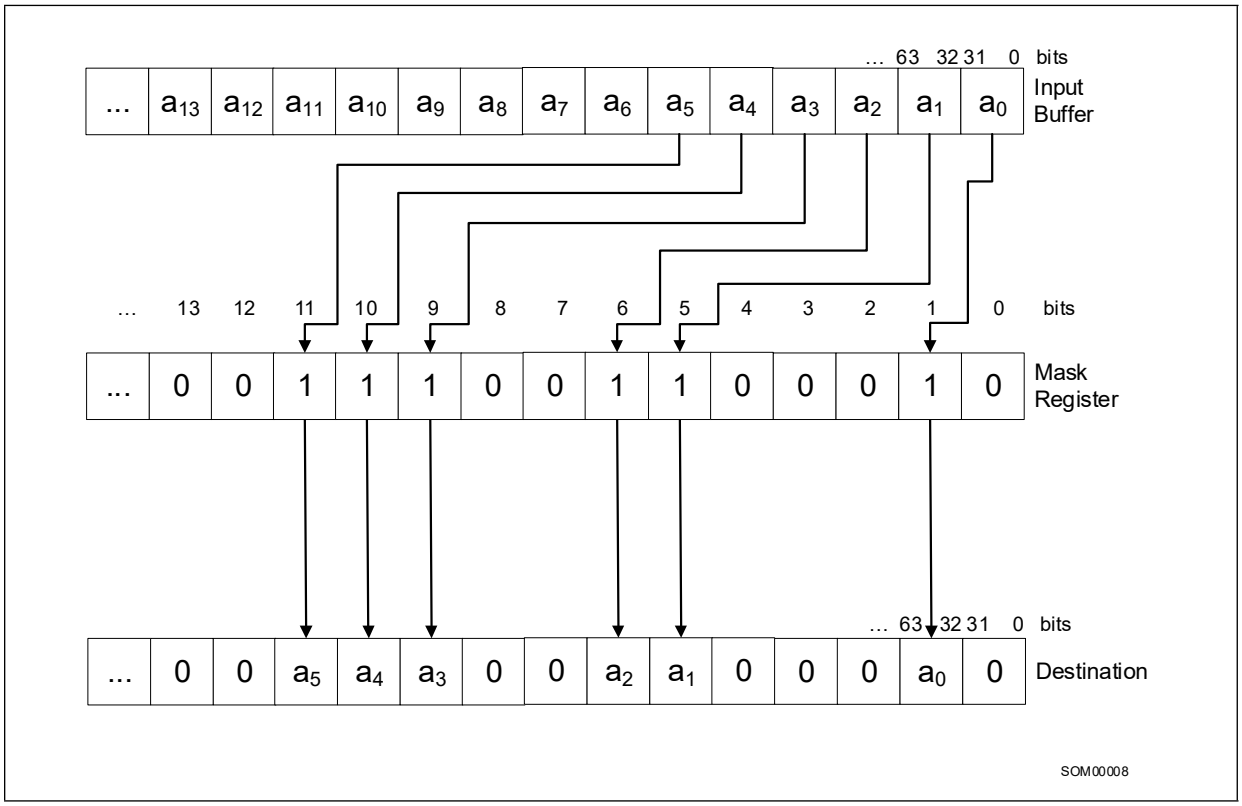


Figure 17-7. Data Expand Operation

17.6.1 DATA EXPAND EXAMPLE

The following snippet shows an example of using the expand operation. For every positive number in an array, the code sets its consecutive number among positives.

```
for (int i=0; i<SIZE; i++)
{
                    if (a[i] > 0)
                    dest[i] = a[count++];
                    else
                    dest[i] = 0;
}
```

Here are three implementations for the expand operation from an array of 16 dword elements.

- **Alternative 1:** uses scalar data access and checks each element separately. If it is greater than 0 then the corresponding element in the destination array is rewritten with the value from source value at index count, and the counter is incremented.
- **Alternative 2:** shows Intel AVX2 code that uses a shuffle instruction together with the preallocated and preinitialized table with shuffle keys.
  - The compare instruction provides the entry point number to the shuffle-key table.
  - The key is then loaded and the original array is shuffled according to the keys.

— Four elements are processed in each iteration.

- **Alternative 3:** shows Intel AVX-512 code, which uses the vpexpandd instruction together with the mask register as an expand key. Sixteen elements are processed in each iteration.

Example 17-14. Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives

Alternative 1: Scalar	Alternative 2: Intel® AVX2	Alternative 3: Intel® AVX-512

mov rsi, input
mov rdi, output
mov r9, len
xor r8, r8
xor r10, r10
mainloop:
mov r11d, dword ptr [rsi+r8*4]
test r11d, r11d
jle m1
mov r11d, dword ptr [rsi+r10*4]
mov dword ptr [rdi+r8*4], r11d
inc r10
m1:
inc r8
cmp r8, r9
jne mainloop
	mov rsi, input
	mov rdi, output
	mov r9, len
	xor r8, r8
	xor r10, r10
	vpxor ymm0, ymm0, ymm0
	mov r14, shuf2
	mainloop:
	vmovdqa ymm1, [rsi+r8*4]
	vpxor ymm4, ymm4, ymm4
	vpcmpgtd ymm2, ymm1, ymm0
	vmovdqu ymm1, [rsi+r10*4]
	vmovmskps r13, ymm2
	shl r13, 5
	vmovdqa ymm3, [r14+r13]
	vpermd ymm4, ymm3, ymm1
	popcnt r13, r13
	add r10, r13
	vmaskmovps [rdi+r8*4], ymm2, ymm4
	add r8, 8
	cmp r8, r9
	jne mainloop
	// The lookup table is too large to
	// reproduce in the document. It
	// consists of 256 rows of 8 32-bit
	// integers. The first 8 and the last 8
	// rows are shown below. The table
	// needs to be 32-byte aligned.
	shuf2:
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 1, 0, 0, 0, 0, 0, 0
	.int 0, 0, 0, 0, 0, 0, 0, 0
	.int 0, 0, 1, 0, 0, 0, 0, 0
	.int 0, 0, 1, 0, 0, 0, 0, 0
	.int 0, 1, 2, 0, 0, 0, 0, 0
		vpxord zmm0, zmm0, zmm0
		mainloop:
		vmovdqa32 zmm1, [rsi+r8*4]
		vpcmpgtd k1, zmm1, zmm0
		vmovdqu32 zmm1, [rsi+r10*4]
		vpexpandd zmm2 {k1}{z}, zmm1
		vmovdqu32 [rdi+r8*4], zmm2
		add r8, 16
		kmovd r11d, k1
		popcnt r12, r11
		add r10, r12
		cmp r8, r9
		jne mainloop

Example 17-14. (Contd.)Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives

Alternative 1: Scalar	Alternative 2: Intel® AVX2	Alternative 3: Intel® AVX-512
	// Skipping 240 lines .int 0, 0, 0, 0, 1, 2, 3, 4 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 1, 0, 2, 3, 4, 5, 6 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 0, 1, 2, 3, 4, 5, 6 .int 0, 0, 1, 2, 3, 4, 5, 6 .int 0, 1, 2, 3, 4, 5, 6, 7	
Baseline 1x	Speedup: 4.23x	Speedup: 8.58x

17.7 TERNARY LOGIC

A ternary logic *vpternlog* operation executes any bitwise logical function between three operands in one instruction. The instruction requires three operands and an immediate value, which is the truth table of this logical expression. The first operand is used as destination, and, therefore, destroyed after the execution.

17.7.1 TERNARY LOGIC EXAMPLE 1

The following example shows a bitwise logic function of three variables. The function in this example is defined by the following truth table.

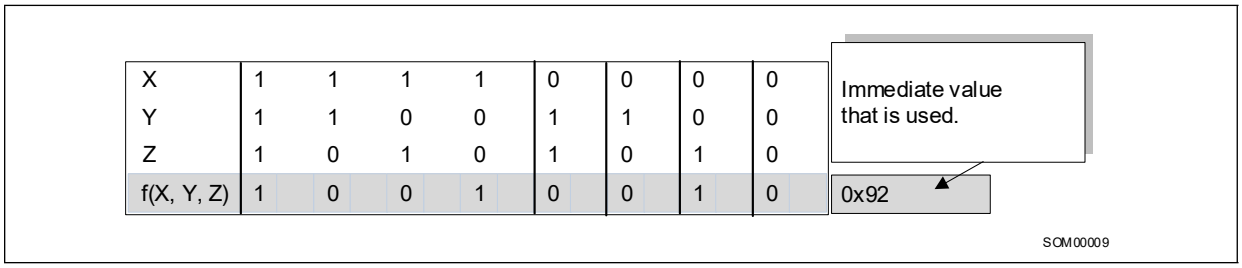


Figure 17-8. Ternary Logic Example 1 Truth Table

Using Karnaugh maps on this truth table, we can define the function as:

f(X,Y,Z) = *y(z ⊕ x)Vxyz*

The C code for the function above is as follows:

```
for (int i=0; i<SIZE; i++)
{
        Dst[i] = ((~Src2[i]) & (Src1[i] ^ Src3[i])) | (Src1[i] & Src2[i] & Src3[i]);
}
```

The value of the function for each combination of X, Y and Z gives an immediate value that is used in the instruction.

Here are three implementations for this logical function applied to all values in X, Y and Z arrays.

- **Alternative 1:** an Intel AVX2 256-bit vector computation, using bitwise logical functions available in Intel AVX2.

- **Alternative 2:** a 512-bit vector computation, using bitwise logical functions available in Intel AVX-512, without using the *vpternlog* instruction.
- **Alternative 3:** an Intel AVX-512 512-bit vector computation, using the *vpternlog* instruction.

All alternatives in the table are unrolled by factor 2.

Example 17-15. Comparing Ternary Logic to Alternative 1

Alternative 1: Intel® AVX2

mov rax, src1
mov rbx, src2
mov rcx, src3
mov r11, dst
mov r8, len
xor r10, r10
mainloop:
vmovdqu ymm1, ymmword ptr [rax+r10*4]
vmovdqu ymm3, ymmword ptr [rdx+r10*4]
vmovdqu ymm2, ymmword ptr [rcx+r10*4]
vmovdqu ymm10, ymmword ptr [rcx+r10*4+0x20]
vpand ymm0, ymm1, ymm3
 	
vpxor ymm4, ymm1, ymm2
vpand ymm5, ymm0, ymm2
vpandn ymm6, ymm3, ymm4
vpor ymm7, ymm5, ymm6
vmovdqu ymmword ptr [r11+r10*4], ymm7
vmovdqu ymm9, ymmword ptr [rax+r10*4+0x20]
vmovdqu ymm11, ymmword ptr [rdx+r10*4+0x20]
vpxor ymm12, ymm9, ymm10
vpand ymm8, ymm9, ymm11
vpandn ymm14, ymm11, ymm12
vpand ymm13, ymm8, ymm10
vpor ymm15, ymm13, ymm14
vmovdqu ymmword ptr [r11+r10*4+0x20], ymm15
 	
add r10, 0x10
cmp r10, r8
jb mainloop

Baseline 1x

Example 17-16. Comparing Ternary Logic to Alternatives 2 and 3

Alternative 2: Intel® AVX-512 Logic Instructions	Alternative 3: Intel® AVX-512 using *vpternlog* Instruction

mov rdi, src1
mov rsi, src2
mov rdx, src3
mov r11, dst
mov r8, len
 	
xor r10, r10
 	
mainloop:
vmovups zmm2, zmmword ptr [rdi+r10*4]
vmovups zmm4, zmmword ptr [rdi+r10*4+0x40]
vmovups zmm6, zmmword ptr [rsi+r10*4]
vmovups zmm8, zmmword ptr [rsi+r10*4+0x40]
vmovups zmm3, zmmword ptr [rdx+r10*4]
vmovups zmm5, zmmword ptr [rdx+r10*4+0x40]
vpandd zmm0, zmm2, zmm6
vpandd zmm1, zmm4, zmm8
vpxord zmm7, zmm2, zmm3
vpxord zmm9, zmm4, zmm5
vpandd zmm10, zmm0, zmm3
vpandd zmm12, zmm1, zmm5
vpandnd zmm11, zmm6, zmm7
vpandnd zmm13, zmm8, zmm9
vpord zmm14, zmm10, zmm11
	mov r9, src1
	mov r8, src2
	mov r10, src3
	mov r11, dst
	mov rsi, len
	
	xor rax rax
	
	mainloop:
	vmovaps zmm1, [r8+rax*4]
	vmovaps zmm0, [r9+rax*4]
	vpternlogd zmm0,zmm1,[r10], 0x92
	vmovaps [r11], zmm0
	vmovaps zmm1, [r8+rax*4+0x40]
	vmovaps zmm0, [r9+rax*4+0x40]
	vpternlogd zmm0,zmm1, [r10+0x40], 0x92
	vmovaps [r11+0x40], zmm0
	addrax, 32
	add r10, 0x80
	add r11, 0x80
	cmp rax, rsi
	jne mainloop

vpord zmm15, zmm12, zmm13
vmovups zmmword ptr [r11+r10*4], zmm14
vmovups zmmword ptr [r11+r10*4+0x40], zmm15
add r10, 0x20
cmp r10, r9
jb mainloop	

Speedup: 1.94x	Speedup: 2.36x (1.22x vs Intel® AVX-512 with logic instructions)

17.7.2 TERNARY LOGIC EXAMPLE 2

The next example is a sign change operation, frequently used in Fortran. Consider the following code, running on two arrays of floating point numbers.

```
for (int i=0; i<SIZE; i++)
{
                b[i] = a[i] > 0 ? b[i] : -b[i];
}
```

This code is equivalent to:

```
for (int i=0; i<SIZE; i++)
{
    b[i] = ( a[i] & 0x80000000 ) ^ b[i];
}
```

Or, in other words:

$$x = (y \wedge z) \oplus x$$

This logic expression gives the following truth table.

X	1	1	1	1	0	0	0	0	Immediate value that is used in the vpternlog instruction.
Y	1	1	0	0	1	1	0	0	
Z	1	0	1	0	1	0	1	0	
f(X, Y, Z)	0	1	1	1	1	0	0	0	0x78

SOM00010

Figure 17-9. Ternary Logic Example 2 Truth Table

Therefore one *vpternlog* instruction can be used instead of using two logic instructions (*vpand* and *vpxor*):

vpternlog x,y,z,0x78

17.8 NEW SHUFFLE INSTRUCTIONS

Intel AVX-512 added 3 new shuffle operations.

- vpermw: a new single source any-to-any word permute.
- permt2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding src register).
- permi2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding control register).

The following figure shows how *vpermi2ps* is used. Notice that in the following example zmm0 is the shuffle control but also the output register (the control register is overridden).

vpermi2ps zmm0, zmm1, zmm2

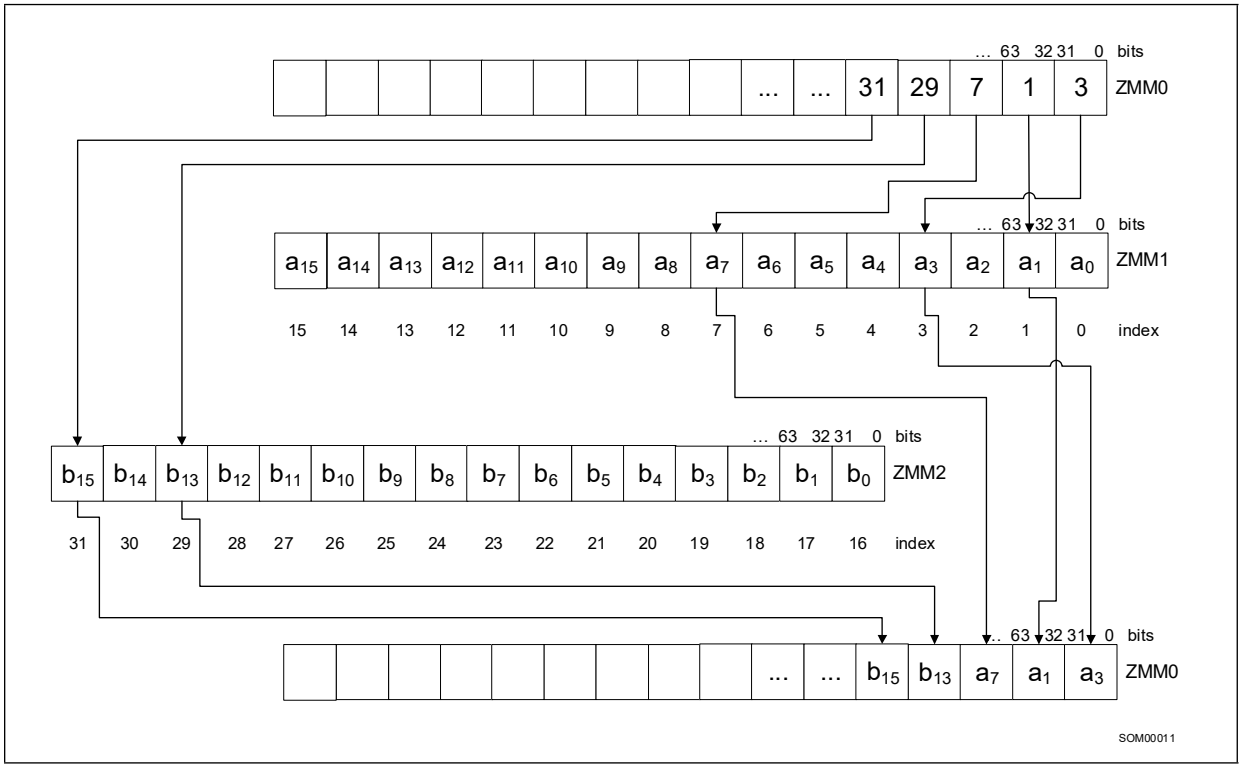


Figure 17-10. VPERMI2PS Instruction Operation

Note that the index register values must have the same resolution as the instruction and source registers (word when working on words, dword when working on dwords, etc.).

17.8.1 TWO SOURCE PERMUTE EXAMPLE

In this example we will show the use of the two source permute instructions in a matrix transpose operation. The matrix we want to transpose is square 8x8 matrix of word elements.

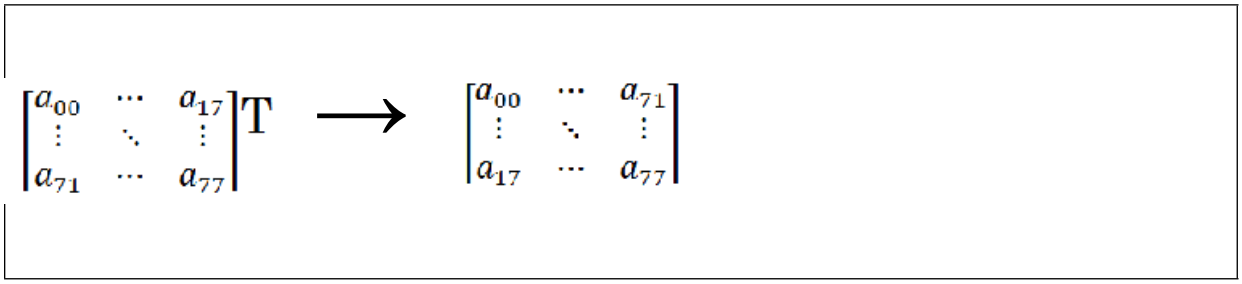


Figure 17-11. Two-Source Permute Instructions in a Matrix Transpose Operation

The corresponding C code is as follows (assuming each matrix occupies a continuous block of 8*8*2 = 128 bytes):

```
for(int iY = 0; iY < 8; iY++)
                                                                    {
for(int iX = 0; iX < 8; iX++)
{
                        trasposedMatrix[iY*8+iX] = originalMatrix[iX*8+iY];
                                                                    }
}
```

Here are three implementations for this matrix transpose.

- Alternative 1 is scalar code, which accesses each element of the source matrix and puts it to the corresponding place in the destination matrix. This code does 64 (8x8) iterations per 1 matrix.
- Alternative 2 is Intel AVX2 code, which uses Intel AVX2 permutation and shuffle (unpack) instructions. Only 1 iteration per 8x8 matrix is required.
- Alternative 3 Intel AVX-512 code which uses the Two Source Permutation instructions. Note that this code first loads permutation masks, and then matrix data. The mask used to perform the permutation is stored in the following array:

```
short permMaskBuffer [8*8] = { 0, 8, 16, 24, 32, 40, 48, 56,
                                 1, 9, 17, 25, 33, 41, 49, 57,
                                2, 10, 18, 26, 34, 42, 50, 58,
                                3, 11, 19, 27, 35, 43, 51, 59,
                                4, 12, 20, 28, 36, 44, 52, 60,
                                5, 13, 21, 29, 37, 45, 53, 61,
                                6, 14, 22, 30, 38, 46, 54, 62,
                                 7, 15, 23, 31, 39, 47, 55, 63 };
```

Each alternative transposes 50 matrixes, 8x8 2-byte elements each.

Example 17-17. Matrix Transpose Alternatives

Alternative 1: Scalar code	Alternative 2: Intel® AVX2 Code	Alternative 3: Intel® AVX-512 Code

mov rsi, pImage
mov rdi, pOutImage
xor rdx, rdx
matrix_loop:
xor rax, rax
outerloop:
xor rbx, rbx
innerloop:
mov rcx, rax
shl rcx, 3
add rcx, rbx
mov r8w, word ptr [rsi+rcx*2]
mov rcx, rbx
shl rcx, 3
add rcx, rax
mov word ptr [rdi+rcx*2], r8w
add rbx, 1
cmp rbx, 8
jne innerloop
add rax, 1
cmp rax, 8
jne outerloop
add rdx, 1
add rsi, 64*2
add rdi, 64*2
cmp rdx, 50
jne matrix_loop
	mov rsi, pImage
	mov rdi, pOutImage
	xor rdx, rdx
	matrix_loop:
	vmovdqa xmm0, [rsi]
	vmovdqa xmm1, [rsi+0x10]
	vmovdqa xmm2, [rsi+0x20]
	vmovdqa xmm3, [rsi+0x30]
	
	vinserti128 ymm0, ymm0, [rsi+0x40], 0x1
	vinserti128 ymm1, ymm1, [rsi+0x50], 0x1
	vinserti128 ymm2, ymm2, [rsi+0x60], 0x1
	vinserti128 ymm3, ymm3, [rsi+0x70], 0x1
	
	vpunpcklwd ymm4,ymm0,ymm1
	vpunpckhwd ymm5,ymm0,ymm1
	vpunpcklwd ymm6,ymm2,ymm3
	vpunpckhwd ymm7,ymm2,ymm3
	
	vpunpckldq ymm0,ymm4,ymm6
	vpunpckhdq ymm1,ymm4,ymm6
	vpunpckldq ymm2,ymm5,ymm7
	vpunpckhdq ymm3,ymm5,ymm7
	
	vpermq ymm0, ymm0, 0xD8
	vpermq ymm1, ymm1, 0xD8
	vpermq ymm2, ymm2, 0xD8
	vpermq ymm3, ymm3, 0xD8
	
	vmovdqa [rdi], ymm0
	vmovdqa [rdi+0x20], ymm1
	vmovdqa [rdi+0x40], ymm2
	vmovdqa [rdi+0x60], ymm3
	add rdx, 1
	add rsi, 64*2
	add rdi, 64*2
	cmp rdx, 50
	jne matrix_loop
		mov rax, permMaskBuffer
		vmovdqa32 zmm10, [rax]
		vmovdqa32 zmm11, [rax+0x40]
		mov rsi, pImage
		mov rdi, pOutImage
		xor rdx, rdx
		matrix_loop:
		vmovdqa32 zmm2, [rsi]
		vmovdqa32 zmm3, [rsi+0x40]
		vmovdqa32 zmm0, zmm10
		vmovdqa32 zmm1, zmm11
		vpermi2w zmm0, zmm2, zmm3
		vpermi2w zmm1, zmm2, zmm3
		vmovdqa32 [rdi], zmm0
		vmovdqa32 [rdi+0x40], zmm1
		
		add rdx, 1
		add rsi, 64*2
		add rdi, 64*2
		cmp rdx, 50
		jne matrix_loop

Baseline 1x	Speedup: 13.7x	Speedup: 37.3x (2.7x vs Intel® AVX2 code)

17.9 BROADCAST

17.9.1 EMBEDDED BROADCAST

Intel AVX-512 introduces embedded broadcast operations, in which a broadcast operation is implied within the syntax of a non-broadcast instruction. A source from memory can be broadcast, that is, repeated, across all the elements of the effective source operand, up to 16 times for a 32-bit data element, and up to 8 times for a 64-bit data element, without using an additional source register. This is useful when we want to reuse the same scalar operand for all the operations in a vector instruction.

Embedded broadcast is only enabled on instructions with an element size of 32 or 64 bits; however, new FP16 instructions allow embedded broadcast. In the case of older technologies, byte and word element broadcasts do not support embedded broadcast. Use a broadcast instruction, rather than embedded broadcast, to broadcast a byte or word.

Using embedded broadcast can reduce the number of registers used in the code, which may be helpful when register pressure exists.

In addition, when using embedded broadcast the load micro-op is in the same instruction as the operation micro-op, and therefore can benefit from micro fusion.

For example, replace the following code:

vbroadcastss zmm3, [rax]

vmulps zmm1, zmm2, zmm3

with:

vmulps zmm1, zmm2, [rax] {1to16}

The {1to16} primitive does the following:

1. Loads one float32 (single precision) element from memory.
2. Replicates it 16 times to form a vector of 16 32-bit floating point elements.

Intel AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

17.9.2 BROADCAST EXECUTED ON LOAD PORTS

In Skylake Server microarchitecture, a broadcast instruction with a memory operand of 32 bits or above is executed on the load ports; it is not executed on port 5 as other shuffles are. Alternative 2 in the following example shows how

executing the broadcast on the load ports reduces the workload on port 5 and increases performance. Alternative 3 shows how embedded broadcast benefits from both executing the broadcast on the load ports and micro fusion.

Example 17-18. Broadcast Executed on Load Ports Alternatives

Alternative 1: 32-bit Load and Register Broadcast	Alternative 2: Broadcast with a 32-bit Memory Operand	Alternative 3: 32-bit Embedded Broadcast

loop:
vmovd xmm0, [rax]
vpbroadcastd zmm0, xmm0
vpaddd zmm2, zmm1, zmm0
vpermd zmm2, zmm3, zmm2
add rax, 0x4
sub rdx, 0x1
jnz loop
	loop:
	vpbroadcastd zmm0, [rax]
	vpaddd zmm2, zmm1, zmm0
	vpermd zmm2, zmm3, zmm2
	add rax, 0x4
	sub rdx, 0x1
	jnz loop
		loop:
		vpaddd zmm2, zmm1, [rax]{1to16}
		vpermd zmm2, zmm3, zmm2
		add rax, 0x4
		sub rdx, 0x1
		jnz loop

Baseline 1x	Speedup: 1.57x	Speedup: 1.9x

The following example shows that on Skylake Server microarchitecture, 16-bit broadcast is executed on port 5 and therefore does not gain from the memory operand broadcast.

Example 17-19. 16-bit Broadcast Executed on Port 5

Alternative 1: 16-bit Load and Register Broadcast	Alternative 2: Broadcast with a 16-bit Memory Operand

loop:
vmovd xmm0, [rax]
vpbroadcastw zmm0, xmm0
vpaddw zmm2, zmm1, zmm0
vpermw zmm2, zmm3, zmm2
add rax, 0x4
sub rdx, 0x1
jnz loop
	loop:
	vpbroadcastw zmm0, [rax]
	vpaddw zmm2, zmm1, zmm0
	vpermw zmm2, zmm3, zmm2
	add rax, 0x2
	sub rdx, 0x1
	jnz loop

Baseline 1x	Speedup: equal to baseline

Notice that embedded broadcast is not supported for 16-bit memory operands.

17.10 EMBEDDED ROUNDING

By default, the Rounding Mode is set by bits 13:14 of the MXCSR register.

Intel AVX-512 introduces a new instruction attribute called Static (per instruction) Rounding Mode (RM) or Rounding Mode override. This attribute allows a specific arithmetic rounding mode to be applied, ignoring the value of the RM bits in the MXCSR. In combination with the rounding-mode, Intel AVX-512 also has an SAE (“suppress-all-exceptions”) attribute, to disable reporting any floating-point exception flag in the MXCSR. SAE is always implied when rounding-mode is enabled.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In this case, vector length is assumed to be the maximal possible vector length (512-bit in case of Intel AVX-512). The table below summarizes the possible static rounding-mode assignments in Intel AVX-512. Note that some instructions already allow the rounding mode to be statically specified via

immediate bits. In such case, the immediate bits take precedence over the embedded rounding mode in the same way as they take precedence over the bits in MXCSR.RM

17.10.1 STATIC ROUNDING MODE

Static rounding mode functions and descriptions are listed below.

Table 17-3. Static Rounding Mode Functions

Function	Description
{rn-sae}	Round to nearest (even) + SAE
{rd-sae}	Round down (toward -infinity) + SAE
{ru-sae}	Round up (toward +infinity) + SAE
{rz-sae}	Round toward zero (Truncate) + SAE

The following code snippet shows a usage example.

Example 17-20. Embedded vs Non-Embedded Rounding

Using Embedded Rounding	Without Embedded Rounding

vaddps zmm7 {k6}, zmm2, zmm4, {ru-sae}	;rax & rcx point to temporary dword values in memory used to load and save (for restoring) MXCSR value
	
	vstmxcsr [rax] ;load mxcsr value to memory
	mov ebx, [rax] ;move to register
	and ebx, 0xFFFF9FFF ;zero RM bits
	or ebx, 0x5F80 ;put {ru} to RM bits and suppress all exceptions
	mov [rcx], ebx ;move new value to the memory
	vldmxcsr [rcx] ;save to MXCSR
	
	vaddps zmm7 {k6}, zmm2, zmm4 ;operation itself
	
	vldmxcsr [rax] ;restore previous MXCSR value

This piece of code would perform the single-precision floating point addition of vectors zmm2 and zmm4 with round-towards-plus-infinity, leaving the result in vector zmm7 using k6 as a conditional writemask. Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

The following are examples of instructions instances where the static rounding-mode is not allowed.

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2 {rd}, 0x00

; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax] {rd}

; instructions with vector length different than maximal vector length (512-bit)
vaddps ymm7 {k6}, ymm2, ymm4 {rd}

; non-floating point instructions
vpaddd zmm7 {k6}, zmm2, zmm4 {rd}
```

17.11 SCATTER INSTRUCTION

This instruction performs a non-continuous store of data (scatter). Given a base address, a set of signed offsets and a data item, the instruction writes each element in the data register to the memory location computed from the base address and corresponding offset. The instruction stores up to 16 elements (8 elements for qword indices) in a doubleword vector or 8 elements in a quadword vector, to the memory locations pointed to by the base address and index vector. Elements are stored only if their corresponding mask bit is one. The figure below describes the following operation.

```
vscatterdpd   [rax + zmm0]{k1} , zmm1
```

In this example, `rax` contains the base address, `zmm0` contains a set of offsets, and `zmm1` contains data to be written.

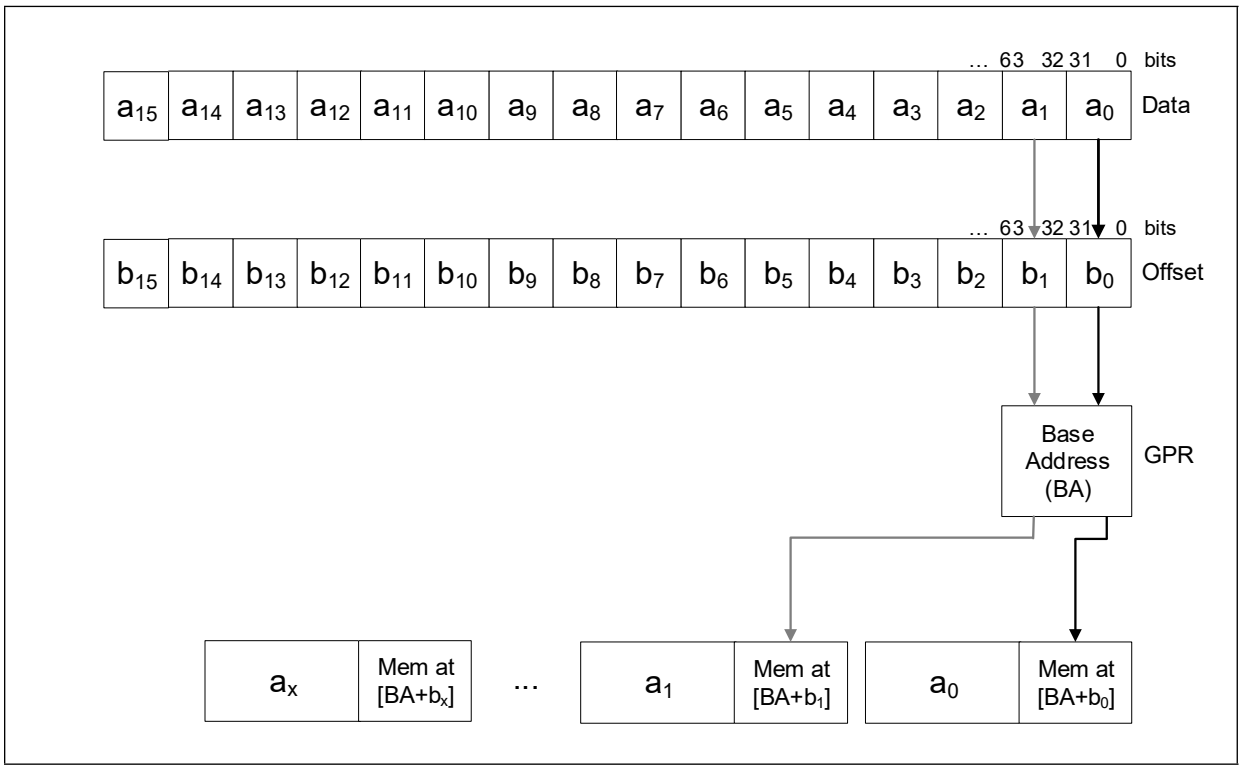


Figure 17-12. VSCATTERDPD Instruction Operation

17.11.1 DATA SCATTER EXAMPLE

Given an array of unique indexes, ranging from 0 to N, we want to sort the array of N values, according to the corresponding index, while converting the values from long long integers (64 bits) to floating point numbers (32 bits).

```
for ( int i=0; i < N; i++ )
{
        dst[ ind [i] ] = (float)src[i];
}
```

Here are three implementations of the code above.

- **Alternative 1:** pure scalar code.
- **Alternative 2:** a software sequence for scatter.
- **Alternative 3:** a hardware scatter.

NOTE

A hardware Scatter operation issues as many store operations, as the number of elements in the vector. Do not use a scatter operation to store sequential elements, which can be stored with one vmov instruction.

Example 17-21. Scalar Scatter

Scalar

mov rax, pImage //input
mov rcx, pOutImage //output
mov rbx, pIndex //indexes
mov rdx, len //length
 	
xor r9, r9
mainloop:
mov r9d, [rbx+rdx-0x4]
vcvtsi2ss xmm0, xmm0, qword ptr [rax+rdx*2-0x8]
vmovss [rcx+r9*4], xmm0
sub rdx, 4
jnz mainloop

Baseline 1x

Table 17-4. Software Sequence and Hardware Scatter

Software Sequence	Hardware Scatter
shufMaskP: .quad·0x0000000200000001 .quad·0x0000000400000003 .quad·0x0000000600000005 .quad·0x0000000800000007 mov rax, pImage //input mov rcx, pOutImage //output mov rbx, pIndex //indexes mov rdx, len //length mov r9, shufMaskP vmovaps ymm2, [r9] mainloop: vmovaps zmm1, [rax + rdx*2 - 0x80] //load data vcvtuqq2ps ymm0, zmm1 //convert to float movsxd r9, [rbx + rdx - 0x40] //load 8th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x3c] //load 7th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x38] //load 6th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x34] //load 5th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0	mov rax, pImage //input mov rcx, pOutImage //output mov rbx, pIndex //indexes mov rdx, len //length mainloop: vmovdqa32 zmm0, [rbx+rdx-0x40] vmovdqa32 zmm1, [rax+rdx*2-0x80] vcvtuqq2ps ymm1, zmm1 vmovdqa32 zmm2, [rax+rdx*2-0x40] vcvtuqq2ps ymm2, zmm2 vshuff32x4 zmm1, zmm1, zmm2, 0x44 kxnorw k1,k1,k1 vscatterdps [rcx+4*zmm0] {k1}, zmm1 sub rdx, 0x40 jnz mainloop

Table 17-4. (Contd.)Software Sequence and Hardware Scatter

Software Sequence	Hardware Scatter
movsxd r9, [rbx + rdx - 0x30] //load 4th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x2c] //load 3rd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x28] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x24] //load 1st index vmovss [rcx + 4*r9], xmm0 vmovaps zmm1, [rax + rdx*2 - 0x40] //load data vcvtuqq2ps ymm0, zmm1 //convert to float movsxd r9, [rbx + rdx - 0x20] //load 8th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x1c] //load 7th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x18] //load 6th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0	
movsxd r9, [rbx + rdx - 0x14] //load 5th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x10] //load 4th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0xc] //load 3rd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x8] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x4] //load 1st index vmovss [rcx + 4*r9], xmm0 sub rdx, 0x40 jnz mainloop	
Speedup: 1.48x	Speedup: 1.53x

17.12 STATIC ROUNDING MODES, SUPPRESS-ALL-EXCEPTIONS (SAE)

The Suppress-all-exceptions (SAE) feature was added to Intel AVX-512 floating-point instructions. This feature is helpful when spurious flag settings are undesirable. Although current implementations of vector math functions usually allow spurious flag settings, they can cause problems for applications that run with exceptions enabled. Standard-compliant code does not allow spurious flag settings.

In addition to standard-mandated uses (IEEE, OpenCL), static rounding modes have applications in math libraries that operate under the default rounding mode (which can be dynamically set).

17.13 QWORD INSTRUCTION SUPPORT

Intel AVX-512 extends QWORD support to many instructions introduced in Intel AVX and Intel AVX2. QWORD support was added to the instructions as detailed in the following sections.

17.13.1 QUADWORD SUPPORT IN ARITHMETIC INSTRUCTIONS

Intel AVX-512 adds new quadword extension to *vpmaxsq, vpmaxuq, vpminsq, vpminuq, and vpmullq*.

The following example will store to array c the max value between the sum and the multiply of two 64bit numbers.

```
const int N = miBufferWidth;
const __int64* restrict a = A;
const __int64* restrict b = B;
__int64* restrict c = Cref;

for (int i = 0; i < N; i++){
        __int64 sum = a[i] + b[i];
        __int64 mul = a[i] * b[i];
        c[i] = mul > sum ? mul : sum;
}
```

The code below shows how the new support reduces instruction count from 118 in Intel AVX2 to 30 in Intel AVX-512 and results in a 3.1x speedup.

Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics

Intel® AVX2 Intrinsics	Intel® AVX-512 Intrinsics

for (int i = 0; i < N; i+= 32){
__m256i aa, bb, aah, bbh, mul, sum;
#pragma unroll(8)
for (int j = 0; j < 8; j++){
aa = _mm256_loadu_si256((const __m256i*)(a+i+4*j));
bb = _mm256_loadu_si256((const __m256i*)(b+i+4*j));
	for (int i = 0; i < N; i+= 32){
	__m512i aa, bb, mul, sum;
	#pragma unroll(4)
	for (int j = 0; j < 4; j++){
	aa = _mm512_loadu_si512((const __m512i*)(a+i+8*j));
	bb = _mm512_loadu_si512((const __m512i*)(b+i+8*j));

Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics (Contd.)

Intel® AVX2 Intrinsics	Intel® AVX-512 Intrinsics

sum = _mm256_add_epi64(aa, bb);
mul = _mm256_mul_epu32(aa, bb);
aah = _mm256_srli_epi64(aa, 32);
bbh = _mm256_srli_epi64(bb, 32);
aah = _mm256_mul_epu32(aah, bb);
bbh = _mm256_mul_epu32(bbh, aa);
aah = _mm256_add_epi32(aah, bbh);
aah = _mm256_slli_epi64(aah, 32);
mul = _mm256_add_epi64(mul, aah);
aah = _mm256_cmpgt_epi64(mul, sum);
aa = _mm256_castpd_si256 (
_mm256_blendv_pd(_mm256_castsi256_pd (sum),
_mm256_castsi256_pd(mul), _mm256_castsi256_pd(
aah)));
_mm256_storeu_si256((__m256i*)(c+4*j), aa);
}
c += 32;
}
	sum = _mm512_add_epi64(aa, bb);
	mul = _mm512_mullo_epi64(aa, bb);
	aa = _mm512_max_epi64(sum, mul);
	_mm512_storeu_si512((__m512i*)(c+8*j), aa);
	
	}
	
	c += 32;
	}

Baseline 1x	Speedup: 3.1x

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
loop: vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0x20] inc r9d vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0x20] vmovdqu32 ymm17, ymmword ptr [r11+rcx*8] vmovdqu32 ymm19, ymmword ptr [rax+rcx*8] vmovdqu ymm13, ymmword ptr [rax+rcx*8+0x40] vmovdqu ymm11, ymmword ptr [r11+rcx*8+0x40] vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm27, ymm26, 0x20 vpsrlq ymm16, ymm19, 0x20 vpsrlq ymm18, ymm17, 0x20 vpaddq ymm6, ymm28, ymm26 vpsrlq ymm10, ymm13, 0x20 vpsrlq ymm12, ymm11, 0x20 vpaddq ymm0, ymm19, ymm17 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpaddd ymm31, ymm29, ymm30 vmovdqu32 ymm29, ymmword ptr [r11+rcx*8+0x80] vpsllq ymm5, ymm31, 0x20 vmovdqu32 ymm31, ymmword ptr [rax+rcx*8+0x80] vpsrlq ymm30, ymm29, 0x20 vpmuludq ymm20, ymm16, ymm17	loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm5, zmmword ptr [rax+rcx*8+0x40] vmovups zmm10, zmmword ptr [rax+rcx*8+0x80] vmovups zmm15, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm6, zmmword ptr [r11+rcx*8+0x40] vmovups zmm11, zmmword ptr [r11+rcx*8+0x80] vmovups zmm16, zmmword ptr [r11+rcx*8+0xc0] vpaddq zmm2, zmm0, zmm1 vpmullq zmm3, zmm0, zmm1 vpaddq zmm7, zmm5, zmm6 vpmullq zmm8, zmm5, zmm6 vpaddq zmm12, zmm10, zmm11 vpmullq zmm13, zmm10, zmm11 vpaddq zmm17, zmm15, zmm16 vpmullq zmm18, zmm15, zmm16 vpmaxsq zmm4, zmm2, zmm3 vpmaxsq zmm9, zmm7, zmm8 vpmaxsq zmm14, zmm12, zmm13 vpmaxsq zmm19, zmm17, zmm18 vmovups zmmword ptr [rsi], zmm4

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly

vpmuludq ymm21, ymm18, ymm19
vpmuludq ymm4, ymm28, ymm26
vpaddd ymm22, ymm20, ymm21
vpaddq ymm7, ymm4, ymm5
vpsrlq ymm28, ymm31, 0x20
vmovdqu32 ymm20, ymmword ptr [r11+rcx*8+0x60]
vpsllq ymm24, ymm22, 0x20
vmovdqu32 ymm22, ymmword ptr [rax+rcx*8+0x60]
vpsrlq ymm21, ymm20, 0x20
vpaddq ymm4, ymm22, ymm20
vpcmpgtq ymm8, ymm7, ymm6
vblendvpd ymm9, ymm6, ymm7, ymm8
vmovups ymmword ptr [rsi+0x20], ymm9
vpmuludq ymm14, ymm10, ymm11
vpmuludq ymm15, ymm12, ymm13
vpmuludq ymm8, ymm28, ymm29
vpmuludq ymm9, ymm30, ymm31
vpmuludq ymm23, ymm19, ymm17
vpaddd ymm16, ymm14, ymm15
vpsrlq ymm19, ymm22, 0x20
vpaddd ymm10, ymm8, ymm9
vpaddq ymm1, ymm23, ymm24
vpsllq ymm18, ymm16, 0x20
vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0xc0]
vpsllq ymm12, ymm10, 0x20
vpmuludq ymm23, ymm19, ymm20
vpmuludq ymm24, ymm21, ymm22
vpaddd ymm25, ymm23, ymm24
vmovdqu32 ymm19, ymmword ptr [rax+rcx*8+0xa0]
vpsllq ymm27, ymm25, 0x20
vpsrlq ymm25, ymm28, 0x20
vpsrlq ymm16, ymm19, 0x20
vpcmpgtq ymm2, ymm1, ymm0
vblendvpd ymm3, ymm0, ymm1, ymm2
vpaddq ymm0, ymm13, ymm11
vmovups ymmword ptr [rsi], ymm3
vpmuludq ymm17, ymm13, ymm11
vpmuludq ymm11, ymm31, ymm29
vpaddq ymm1, ymm17, ymm18
vpaddq ymm13, ymm31, ymm29
vpaddq ymm14, ymm11, ymm12
vmovdqu32 ymm17, ymmword ptr [r11+rcx*8+0xa0]
vmovdqu ymm12, ymmword ptr [r11+rcx*8+0xe0]
vpsrlq ymm18, ymm17, 0x20
vpcmpgtq ymm2, ymm1, ymm0
vpmuludq ymm26, ymm22, ymm20
vpcmpgtq ymm15, ymm14, ymm13
vblendvpd ymm3, ymm0, ymm1, ymm2
vblendvpd ymm0, ymm13, ymm14, ymm15
vmovdqu ymm14, ymmword ptr [rax+rcx*8+0xe0]
	vmovups zmmword ptr [rsi+0x40], zmm9
	vmovups zmmword ptr [rsi+0x80], zmm14
	vmovups zmmword ptr [rsi+0xc0], zmm19
	add rcx, 0x20
	add rsi, 0x100
	cmp r9d, r8d
	jb loop

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly

vmovups ymmword ptr [rsi+0x40], ymm3
vmovups ymmword ptr [rsi+0x80], ymm0
vpaddq ymm5, ymm26, ymm27
vpsrlq ymm11, ymm14, 0x20
vpsrlq ymm13, ymm12, 0x20
vpaddq ymm1, ymm19, ymm17
vpaddq ymm0, ymm14, ymm12
vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0xc0]
vpmuludq ymm20, ymm16, ymm17
add rcx, 0x20
vpmuludq ymm21, ymm18, ymm19
vpaddd ymm22, ymm20, ymm21
vpsrlq ymm27, ymm26, 0x20
vpsllq ymm24, ymm22, 0x20
vpmuludq ymm29, ymm25, ymm26
vpmuludq ymm30, ymm27, ymm28
vpmuludq ymm15, ymm11, ymm12
vpmuludq ymm16, ymm13, ymm14
vpmuludq ymm23, ymm19, ymm17
vpaddd ymm31, ymm29, ymm30
vpaddd ymm17, ymm15, ymm16
vpaddq ymm2, ymm23, ymm24
vpsllq ymm19, ymm17, 0x20
vpcmpgtq ymm6, ymm5, ymm4
vblendvpd ymm7, ymm4, ymm5, ymm6
vpsllq ymm6, ymm31, 0x20
vmovups ymmword ptr [rsi+0x60], ymm7
vpaddq ymm7, ymm28, ymm26
vpcmpgtq ymm3, ymm2, ymm1
vpmuludq ymm5, ymm28, ymm26
vpmuludq ymm18, ymm14, ymm12
vblendvpd ymm4, ymm1, ymm2, ymm3
vpaddq ymm8, ymm5, ymm6
vpaddq ymm1, ymm18, ymm19
vmovups ymmword ptr [rsi+0xa0], ymm4
vpcmpgtq ymm9, ymm8, ymm7
vpcmpgtq ymm2, ymm1, ymm0
vblendvpd ymm10, ymm7, ymm8, ymm9
vblendvpd ymm3, ymm0, ymm1, ymm2
vmovups ymmword ptr [rsi+0xc0], ymm10
vmovups ymmword ptr [rsi+0xe0], ymm3
add rsi, 0x100
cmp r9d, r8d
jb loop	

Baseline 1x	Speedup: 3.1x

17.13.2 QUADWORD SUPPORT IN CONVERT INSTRUCTIONS

The following tables demonstrate the new quadword extension in convert instructions.

Table 17-5. Vector Quadword Extensions

From / To	Vector SP	Vector DP	Vector int64	Vector uint64
Vector SP	-		vcvtps2qq	vcvtps2uqq
Vector DP		-	vcvtpd2qq	vcvtpd2qq
Vector int64	vcvtqq2ps	vcvtqq2pd	-	
Vector uint64	vcvtqq2ps	vcvtuqq2pd		-

Table 17-6. Scalar Quadword Extensions

From / To	Scalar SP	Scalar DP	Scalar int64	Scalar uint64
Scalar SP	-	-	vcvtss2si	vcvtss2usi
Scalar DP	-	-	vcvtsd2si	vcvtsd2usi
Scalar int64	vcvtsi2sd	vcvtsi2sd	-	-
Scalar uint64	vcvtusi2sd	vcvtusi2sd	-	-

17.13.3 QUADWORD SUPPORT FOR CONVERT WITH TRUNCATION INSTRUCTIONS

The following tables demonstrate the new quadword extension in convert with truncate instructions.

Table 17-7. Vector Quadword Extensions

From / To	Vector int64	Vector uint64
Vector SP	vcvttps2qq	vcvttps2uqq
Vector DP	vcvttpd2qq	vcvttpd2qq

Table 17-8. Scalar Quadword Extensions

From / To	Scalar int64	Scalar uint64
Scalar SP	vcvttss2si	vcvttss2usi
Scalar DP	vcvttsd2si	vcvttsd2usi

17.14 VECTOR LENGTH ORTHOGONALITY

All Intel AVX-512 instructions, in processors that support Vector Length Extensions (VL), can operate at three vector lengths: 128-bit, 256-bit and 512-bit. All of these vector lengths are supported by all Intel AVX-512 instructions, except instructions with Embedded Rounding.

In the instruction encoding, the same two bits are used for encoding vector length and embedded rounding control, therefore when embedded rounding is used, the vector length is automatically assumed to be 512 bits (maximum vector length in Intel AVX-512).

See also Section 17.10.

17.15 INTEL® AVX-512 INSTRUCTIONS FOR TRANSCENDENTAL SUPPORT

This section lists and describes the new instructions introduced by Intel AVX-512 for transcendental support.

17.15.1 VRCP14, VRSQRT14 - SOFTWARE SEQUENCES FOR 1/X, X/Y, SQRT(X)

Syntax:

VRCP14PD/PS dest, src

VRSQRT14PD/PS dest, src

17.15.1.1 Application Examples

There are software sequences for Reciprocal, Division, Square Root, and Inverse Square Root instructions.

Software sequences for 1/x, x/y, sqrt(x) are beneficial for throughput (not so much for latency, unless the accuracy is quite low). They are typically implemented via Newton-Raphson approximations, or polynomial approximations.

One advantage of VRCP14 and VRSQRT14 is the improved accuracy, compared with the legacy RCPPS, RSQRTPS. This helps shorten the computation, in particular for double precision (which requires two instead of three Newton-Raphson iterations for a 50-52 bit approximation).

Another advantage of these instructions is that they have double-precision versions (while the legacy RCP/RSQRT instructions did not). This further boosts double-precision performance. On Skylake Server microarchitecture, double precision reciprocal and square root software sequences have significantly better throughput than the VDIV and VSQRT instructions in 512-bit vector mode Double Precision Transcendental Argument Reductions (e.g., log, cbrt).

In functions such as log() or the cube root (cbrt), a rounded VRCP14PD result can be used in place of an expensive reciprocal table lookup. The same technique could be used before via RCPPS, but was less efficient for double-precision.

See Section 17.15.3 for a log() argument reduction example.

17.15.2 VGETMANT VGETEXP - VECTOR GET MANTISSA AND VECTOR GET EXPONENT

Syntax:

VGETMANTPD/PS dest_mant, src, imm

VGETEXPPD/PS dest_exp, src

17.15.2.1 Application Examples

Logarithm Function

log2(x) = VGETEXP(x) + log2(VGETMANT(x,8))

log(x) = VGETEXP(x)*log(2.0) + log(VGETMANT(x,8))

As seen above, the computation is reduced to computing log(VGETMANT(x,8)), where VGETMANT(x,8) is guaranteed to be in [1,2) for all valid function inputs, and NaN for invalid inputs (x<0).

A variety of algorithms can be applied to compute the logarithm of the mantissa. The selection of a particular algorithm may depend on the desired accuracy, on optimization goals (latency or throughput optimized), or on specifics of the microarchitecture. Some algorithms may use other normalization options for the mantissa: [0.5, 1) or [0.75, 1.5); however, the basic identity underlying the computation is shown above.

See Section 17.15.5 for details on X^{alpha} (constant alpha) and division.

17.15.3 VRNDSCALE - VECTOR ROUND SCALE

Syntax:

VRNDSCALEPD/PS dest, src, imm

17.15.3.1 Application Examples

Lookup tables are frequently used in transcendental function implementations. The table index is most often based on a few leading bits of the input. VRNDSCALE can be used as part of the argument reduction process, to form the floating-point input value corresponding to the table index. The following example implements the argument reduction for log(x), where $1 \leq x < 2$:

y = RCP14(x); // y is in (0.5, 1]

y0=RNDSCALE(y, k*16); // y0 has k mantissa bits (leading 1

// included)

R = x?y0 - 1; // |R|<2-14+2-k.

Therefore log(x) = -log(y0) + log(1+R).

log(1+R)can be computed via a polynomial, and log(y0) can be retrieved from a lookup table of 2k-1+1 elements, or 2k-1 elements, at the expense of an additional check.

17.15.4 VREDUCE - VECTOR REDUCE

Syntax:

VREDUCEPD/PS dest, src, imm

17.15.4.1 Application Examples

The most significant benefit of VREDUCE is latency reduction in common transcendental operations such as exp2 and pow (which includes an exp2 operation). Uses in other transcendental functions such as atan() are also possible.

17.15.5 VSCALEF - VECTOR SCALE

Syntax:

VSCALEFPD/PS dest, src1, src2

17.15.5.1 Application Examples

exp2 (2x)

exp2(x) = VSCALEF(2VREDUCE(x, RD_mode), x)

R(x) = VREDUCE(x, RD_mode) = x - floor(x) is in [0, 1). 2R(x) is computed by other means, such as polynomial approximation, or table lookup with polynomial approximation. VSCALEF correctly handles overflow and underflow. It is also defined to handle exp() special cases correctly (such as when the input is an Infinity), so there is no need for special paths in a vector implementation. In the absence of VSCALEF, inputs that are very large in magnitude require a separate path.

Since explicit exponent manipulation is no longer needed, VSCALEF also helps improve throughput.

Exp(x)

Exp(x) = VSCALEF(2R(x), x*(1/log(2.0)),

where,

R(x) = x - log(2.0)*floor(x*(1/log(2.0));

R(x) is accurately computed by using a sufficiently long log(2.0) approximation (longer than the native floating-point format).

As with exp2(), the advantages of using VSCALEF are better throughput and elimination of secondary branches.

x^{alpha} (constant alpha)

For example, alpha=1/3 (the cube root function, cbrt).

The basic reduction for this computation is:

x^{alpha} = VSCALEF((VGETMANT(x, imm))alpha?2VREDUCE (VGETEXP(x)*alpha, RD_mode), VGETEXP(x)*alpha)

selecting the immediate (imm) is based on the value of the alpha constant.

Division:

a/b = VSCALEF(VGETMANT(a,0)/VGETMANT(b,0), VGETEXP(a)-VGETEXP(b))

This reduction allows for a branch-free implementation of divide, that covers overflow, underflow, and special inputs (zeroes, Infinities, or denormals).

|VGETMANT(x,0)| is in [1,2) for all non-NaN inputs.

VGETMANT(a,0)/VGETMANT(b,0) can be computed to the desired accuracy.

The suppress-all-exceptions (SAE) feature available in Intel AVX-512 can help ensure spurious flag settings do not occur. Flags can be set correctly as part of the computation (except for divide-by-zero, which requires an additional step).

For high accuracy or IEEE compliance, the hardware instruction typically provides better performance, especially in terms of latency.

17.15.6 VFPCLASS - VECTOR FLOATING POINT CLASS

Syntax:

VFPCLASSPD/PS dest_mask, src, imm

17.15.6.1 Application Examples

The VFPCLASS instruction is used to detect special cases so they can be directed to a special path, or alternatively, handled with masked operations in the main path. See two examples below.

Reciprocal Sequence, Square Root Sequence:

- The reduced argument for the 1/x computation is e=1-x*RCP14(x).
  - This expression evaluates to NaN when x is ±0 or ±Inf, as RCP14 returns the correct result for these special cases.
  - VFPCLASS enables you to set mask=1 for x=±0 or ±Inf, and mask=0 for all other x.
    - This mask can then be used to select between the RCP14 output (result for special cases), or the result of a reciprocal refinement computation starting with RCP14 (for typical inputs).
- In a similar manner, a square root computation based on RSQRT14 can use the VFPCLASS instruction to create a mask for =±0 or x=+Inf.

Pow(x,y) function:

- The main path of pow(x,y)=2y*log2(x) does not operate on x?0, x=Inf/NaN, or y=Inf/NaN.
- One VFPCLASS op can be used to set special_x_mask=1 for x?0 or x=Inf/NaN.
- A second VFPCLASS op would be used to set special_y_mask=1 for y=Inf/NaN.
- A branch to a secondary path is taken if either mask is set.

17.15.7 VPERM, VPERMI2, VPERMT2 - SMALL TABLE LOOKUP IMPLEMENTATION

17.15.7.1 Application Examples

Math library functions are frequently implemented using table lookups. In vector mode, large table lookups would use vector gather. Small table lookups can be implemented via the VPERM* instructions, which are significantly faster.

Examples of common transcendental functions that achieved very significant speedup using VPERM* for table lookups: exp(), log(), pow() - both single and double precision.

17.16 CONFLICT DETECTION

The Intel AVX-512 Conflict Detection instructions are instructions that, together with Intel AVX-512 Foundation instructions, enable efficient vectorization of loops with possible vector dependencies (i.e., conflicts) through memory. VPCONFLICT performs horizontal comparisons of elements within a single vector register. VPCONFLICT compares each element of a vector register with all previous elements in that register, and outputs the results of all of the comparisons. These horizontal comparisons can be used for other purposes.

Other conflict detection instructions allow for efficient manipulation of the comparison results. The VPLZCNT instruction lets us generate controls for in-register permute operations used to combine vector elements with matching values.

17.16.1 VECTORIZATION WITH CONFLICT DETECTION

The Intel AVX-512CD instructions allow efficient vectorization of loops with reads and writes through an array of pointers

For example:

***ptr[i] += val[i]) or an indirectly addressed array (e.g., A[B[i]] += val[i]).**

Consider the following histogram computation:

```
for(int i = 0; i < num_inputs; i++)
                                                    {
                                    histogram[input[i] & (num_bins - 1)]++;
                                                    }
```

If input[0] = input[1] = 3, we will get an incorrect answer if we use SIMD instructions to read histogram[input[0]] and histogram[input[1]] into a register (with a gather), increment them, and then write them back (with a scatter). After this sequence, the value in histogram[3] will be 1, when it should be 2.

The problem occurs because we have duplicate indices; this creates a dependence between the write to the histogram in iteration 0 and the read from the histogram in iteration 1 - the read should get the value of the previous write.

To detect this scenario, look for duplicate indices (or pointer values), using the VPCONFLICT instruction. This instruction compares each element of a vector register with all previous elements in that register.

Example:

vpconflictd zmm0, zmm1

The figure below is an example that shows the execution of a VPCONFLICTD instruction. The input, ZMM1, contains 16 integers, shown in the light grey boxes. ZMM1 is at the top of the figure, and also visually transposed along the left-hand side of the figure. The white boxes show the equality comparisons that the hardware performs between different elements of ZMM1, and the outcome of each comparison (0 = not equal, 1 = equal). Each comparison output is a single bit in the output of the instruction. Comparisons that are not performed (i.e., the dark grey boxes) produce a single '0' bit in the output. Finally, the output register, ZMM0, is shown at the bottom of the figure. Each element is shown as a decimal representation of the bits above it.

Use VPCONFLICT in different ways to help vectorize loops.

The simplest option is to check for any duplicate indices in a given SIMD register. If there are none, SIMD instructions can be used to compute all elements simultaneously. If conflicts are present, execute a scalar loop for that group of elements.

Branching to a scalar version of the loop on any duplicate indices can work well if duplicates are extremely rare. However, if the chance of getting even one duplicate in a given iteration of the vectorized loop is large enough, then it is better to use SIMD as much as possible, to exploit as much parallelism as possible.

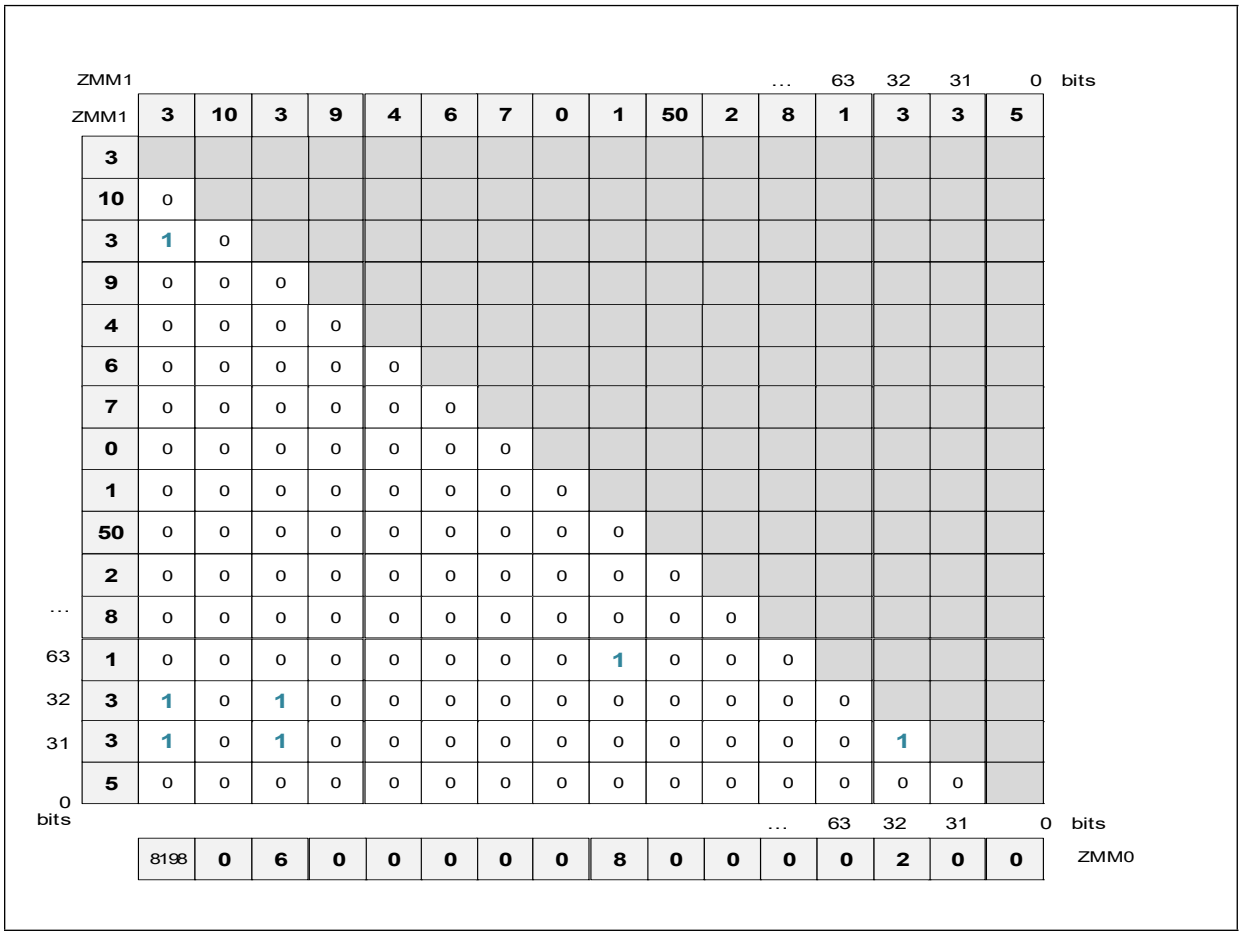


Figure 17-13. VPCONFLICTD Instruction Execution

For loops performing updates to memory locations, such as in the histogram example, minimize store-load forwarding by merging the updates to each distinct index while the data is in registers, and only perform a single write to each memory location. Further, the merge can be performed in a parallel fashion.

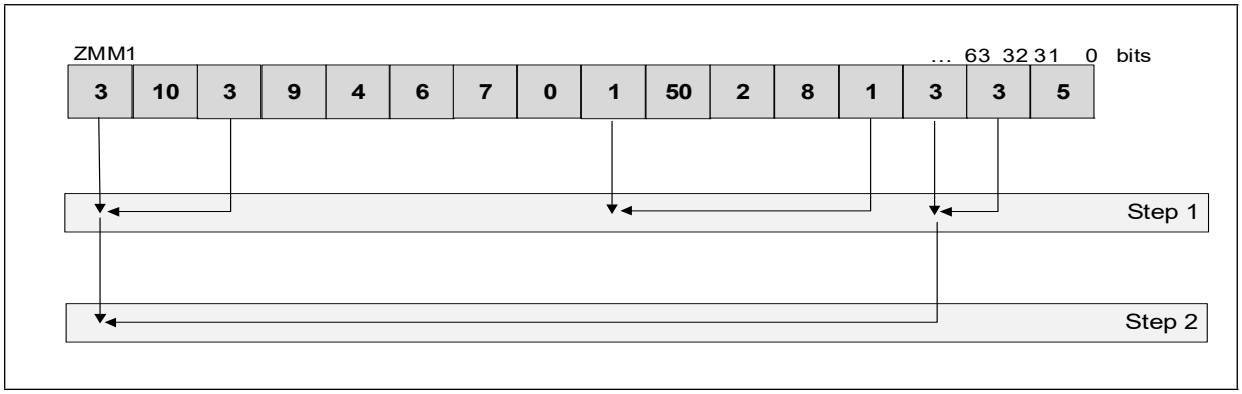


Figure 17-14. VPCONFLICTD Merging Process

The figure above shows the merging process for the example set of indices. While the figure shows only the indices, it actually merges the values. Most of the indices are unique, and thus require no merging. Step 1 combines three pairs of indices: two pairs of '3's and one pair of '1's. Step 2 combines the intermediate results for the '3's from step 1, so that there is now a single value for each distinct index. Notice that in only two steps, the four elements with an index value of 3 are merged, because we performed a tree reduction; we merged pairs of results or intermediate results at each step.

The merging (combining or reduction) process shown above is done with a set of permute operations. The initial permute control is generated with a VPLZCNT+VPSUB sequence. VPLZCNT provides the number of leading zeros for each vector element (i.e., contiguous zeros in the most significant bit positions). Subtracting the results of VPLZCNT from the number of bits in each vector element, minus one, provides the bit position of the most significant '1' bit in the result of the VPCONFLICT instruction, or results in a '-1' for an element if it has no conflicts. In the example above this sequence results in the following permute control.

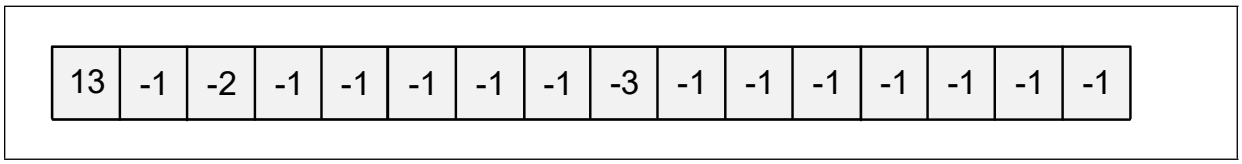


Figure 17-15. VPCONFLICTD Permute Control

The permute loop for merging matching indices and generating the next set of permute indices repeats until all values in the permute control become equal to ‘-1’.

The assembly code below shows both the scalar version of a histogram loop, and the vectorized version with a tree reduction. Speedups are modest because the loop contains little computation; the SIMD benefit comes almost entirely from vectorizing just the logical AND operation and the increment. SIMD speedups can be much higher for loops containing more vectorizable computation.

Example 17-24. Scatter Implementation Alternatives

Scalar Code (Unrolled Two Times)	Intel® AVX-512 Code

mov r9d, bins_minus_1
mov ebx, num_inputs
mov r10, pInput
mov r15, pHistogram
xor rax, rax
histogram_loop:
lea ecx, [rax + rax]
inc eax
movsxd rcx, ecx
mov esi, [r10+rcx*4]
and esi, r9d
mov r8d, [r10+rcx*4+4]
movsxd rsi, esi
and r8d, r9d
movsxd r8, r8d
inc dword ptr [r15+rsi*4]
inc dword ptr [r15+r8*4]
cmp eax, ebx
jb histogram_loop
	vmovaps zmm4, all_1 // {1, 1, …, 1}
	vmovaps zmm5, all_negative_1
	vmovaps zmm6, all_31
	vmovaps zmm7, all_bins_minus_1
	mov ebx, num_inputs
	mov r10, pInput
	mov r15, pHistogram
	xor rcx, rcx
	histogram_loop:
	vpandd zmm3, zmm7, [r10+rcx*4]
	vpconflictd zmm0, zmm3
	kxnorw k1, k1, k1
	vmovaps zmm2, zmm4
	vpxord zmm1, zmm1, zmm1
	vpgatherdd zmm1{k1}, [r15+zmm3*4]
	vptestmd k1, zmm0, zmm0
	kortestw k1, k1
	je update
	
	vplzcntd zmm0, zmm0
	vpsubd zmm0, zmm6, zmm0
	
	conflict_loop:
	vpermd zmm8{k1}{z}, zmm0, zmm2
	vpermd zmm0{k1}, zmm0, zmm0
	vpaddd zmm2{k1}, zmm2, zmm8
	vpcmpned k1, zmm5, zmm0
	kortestw k1, k1
	jne conflict_loop
	
	update:
	vpaddd zmm0, zmm2, zmm1
	kxnorw k1, k1, k1
	add rcx, 16
	vpscatterdd [r15+zmm3*4]{k1}, zmm0
	cmp ecx, ebx
	jb histogram_loop

Scalar, Baseline, 1x	Speedup: 1.11x (random inputs); 1.34x (input values identical)

Notice that the end result of the conflict loop (i.e., the resulting vector after all merging is done, ZMM2 in the above sequence) holds the complete set of partial sums. That is, for each element, the result contains the value of that element merged with all earlier elements with the same index value. Using the earlier example values, ZMM2 contains the result shown in Figure 17-16.

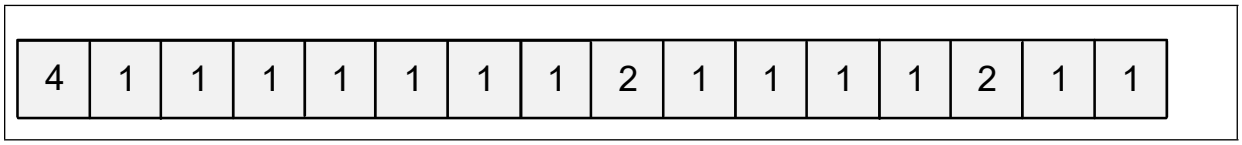


Figure 17-16. VPCONFLICTD ZMM2 Result

While the above sequence does not take advantage of this, other use cases might.

17.16.2 SPARSE DOT PRODUCT WITH VPCONFLICT

A sparse vector may be stored as a pair of arrays: one containing non-zero values, and one containing the original locations of those values in the vector. Note that the indices are sorted in increasing order.

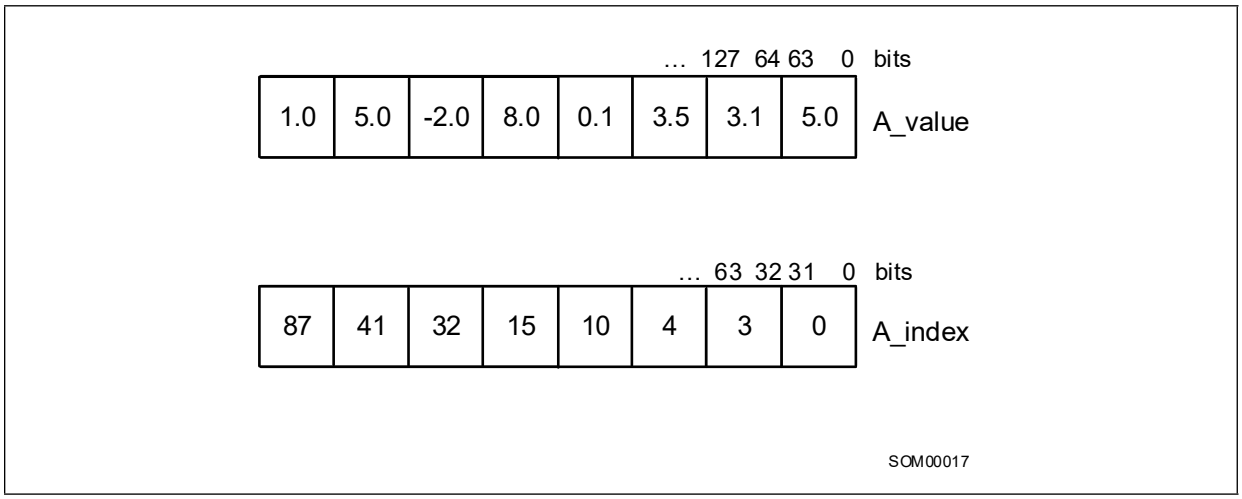


Figure 17-17. Sparse Vector Example

To perform a dot product of two sparse vectors efficiently, we need to find elements with matching indices; those are the only ones on which we should perform the multiply and accumulation. The scalar method for doing this is to start at the beginning of the two index arrays, compare those indices, and if there is a match, do the multiply and accumulate, then advance the indices of both vectors. If there is no match, we advance the index of the lagging vector.

```
A_offset = 0; B_offset = 0; sum = 0;
while ((A_offset < A_length) && (B_offset < B_length))
                                                    {
                                  if (A_index[A_offset] == B_index[B_offset]) // match
                                                    {
                                  sum += A_value[A_offset] * B_value[B_offset];
                                                    A_offset++;
                                                    B_offset++;
                                                    }
```

```
else if (A_index[A_offset] < B_index[B_offset])
{
A_offset++;
}
else
{
B_offset++;
}
}
```

The Intel AVX-512CD instructions provide an efficient way to vectorize this loop. Instead of comparing one index from each vector at a time, we can compare eight of them. First we combine eight indices from each vector into a single vector register. Then, the VPCONFLICT instruction compares the indices. We use the output to create a mask of elements in vector A that have a match, and also to create permute controls to move the corresponding values of B to the same location, so that we can use a vector FMA instruction.

Example 17-25 shows the assembly code for both the scalar and vector versions of a single comparison and FMA. For brevity, the offset updates and looping are omitted.

Example 17-25. Scalar vs. Vector Update Using AVX-512CD

Scalar Code	Intel® AVX-512 Code

mov rdx, A_index
mov rcx, A_offset
mov rax, A_value
mov r12, B_index
mov r13, B_offset
mov rbx, B_value
 	
mov r10d, [rdx+rcx*4]
mov r11d, [r12+r13*4]
cmp r10d, r11d
jne skip_fma
 	
// do the fma on a match
movsd xmm5, [rbx+r13*8]
mulsd xmm5, [rax+rcx*8]
addsd xmm4, xmm5
skip_fma:
	mov rdx, A_index
	mov rcx, A_offset
	mov rax, A_value
	mov r12, B_index
	mov r13, B_offset
	mov rbx, B_value
	mov r14, all_31s // array of {31, 31, …}
	vmovaps zmm2, [r14]
	mov r15, upconvert_control // array of {0, 7, 0, 6, 0, 5, 0, 4, 0, 3, 0, 2, 0, 1, 0, 0}
	vmovaps zmm1, [r15]
	vpternlogd zmm0, zmm0, zmm0, 255
	movl esi, 21845
	kmovw k1, esi // odd bits set
	/ read 8 indices for A
	vmovdqu ymm5, [rdx+rcx*4]
	// read 8 indices for B, and put
	// them in the high part of zmm6
	vinserti64x4 zmm6, zmm5, [r12+r13*4], 1
	vpconflictd zmm7, zmm6
	// extract A vs. B comparisons
	//vextracti64x4 ymm8, zmm7, 1
	// convert comparison results to

Example 17-25. Scalar vs. Vector Update Using AVX-512CD (Contd.)

Scalar Code	Intel® AVX-512 Code

	// permute control
	vplzcntd zmm9, zmm8
	vptestmd k2, zmm8, zmm0
	vpsubd zmm10, zmm2, zmm9
	// upconvert permute controls from
	// 32b to 64b, since data is 64b
	vpermd zmm11{k1}, zmm1, zmm10
	// Move A values to corresponding
	// B values, and do FMA
	vpermpd zmm12{k2}{z}, zmm11, [rax+rcx*8]
	vfmadd231pd zmm4, zmm12, [rbx+r13*8]

Baseline, 1x	Speedup, 4.4x

17.17 INTEL® AVX-512 VECTOR BYTE MANIPULATION INSTRUCTIONS (VBMI)

Intel® AVX-512 VBMI instructions are a set of 512-bit instructions that are designed to speed up bit manipulation operations. The following sections describe the new instructions and show simple usage examples.

See Chapter 15, "Programming with Intel® AVX-512" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for complete instruction definitions.

Processors that provide VBMI1 and VBMI2 are enumerated by the CPUID feature flags CPUID:(EAX=07H, ECX=0):ECX[bit 01] = 1 and CPUID:(EAX=07H, ECX=0):ECX[bit 06] = 1, respectively.

17.17.1 PERMUTE PACKET BYTES ELEMENTS ACROSS LANES (VPERMB)

The VPERMB instruction is a single source, any-to-any byte permute instruction. The following figure shows a VPERMB instruction operation example.

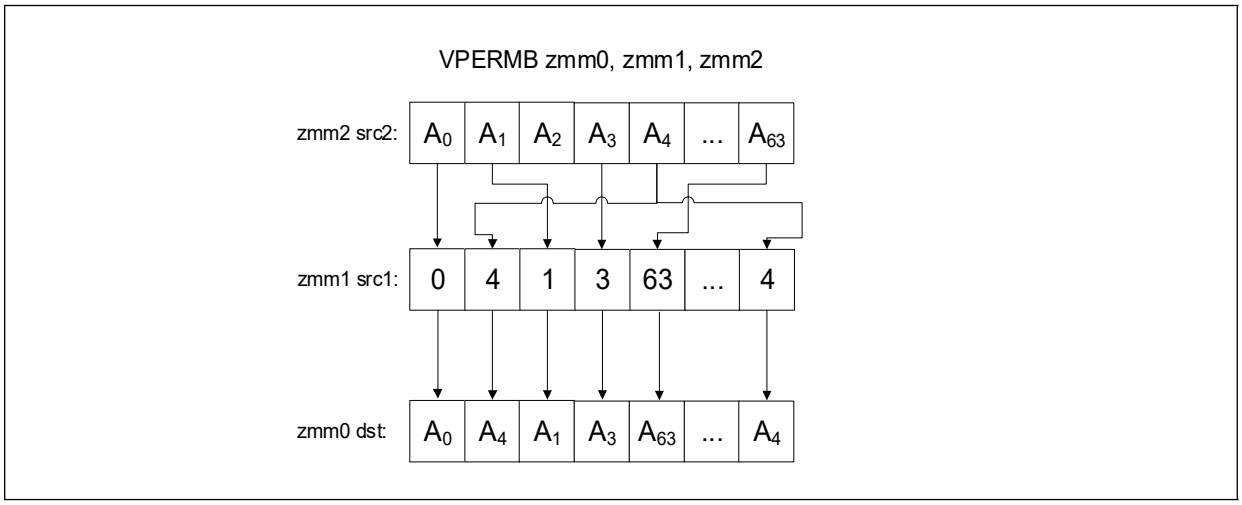


Figure 17-18. VPERMB Instruction Operation

VPERMB Operation:

```
// vpermb zmm Dst {k1}, zmm Src1, zmm Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;

for(int i=0;i<64;i++){
        if(k1[i]){
            Dst[i]= Src2[Src1[i]];
        }else{
            Dst[i]= zero_masking? 0 : Dst[i];
        }
}
```

The following example shows a 64-byte lookup table implementation.

Scalar code:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int
numOfElements){
        for(int i = 0; i < numOfElements; i++) {
                out_bytes[i] = dictionary_bytes[in_bytes[i] & 63];
          }
}
```

Example 17-26. Improvement with VPERMB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMB Implementation

mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
vpmovzxbw zmm3, [rsi]
vpmovzxbw zmm4, [rsi+32]
 	
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpmovzxbw zmm2, [r11+r8*1+32]
vpermi2w zmm1, zmm3, zmm4
vpermi2w zmm2, zmm3, zmm4
vpmovwb [rax+r8*1], zmm1
vpmovwb [rax+r8*1+32], zmm2
add r8, 64
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermb zmm1, zmm1, zmm2
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

Base Measurement: 1x	Speedup: 6.5x

17.17.2 TWO-SOURCE BYTE PERMUTE ACROSS LANES (VPERMI2B, VPERMT2B)

The VPERMI2B and VPERMT2B instructions are two-source byte, permute instructions. The destination is also an operation source; in VPERMI2B the destination is the operation index, and in VPERMT2B the destination is one of the data sources.

The Figure 17-19 shows a VPERMI2B instruction operation example.

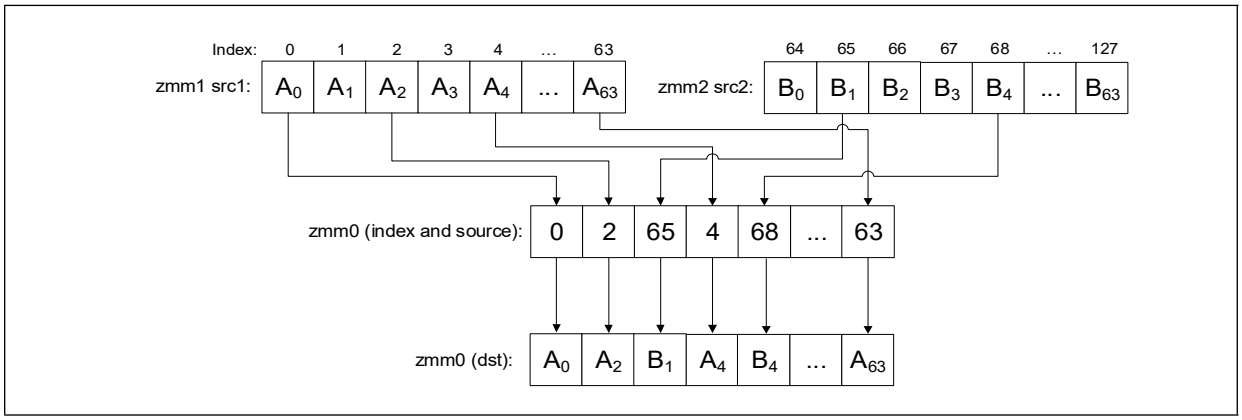


Figure 17-19. VPERMI2B Instruction Operation

VPERMI2B Operation:

```
/// vpermi2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;
for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Dst [i]>63 ? Src1[Dst [i] & 63] : Src2[Dst [i] & 63]  ;
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

The following figure shows a VPERMT2B instruction operation example.

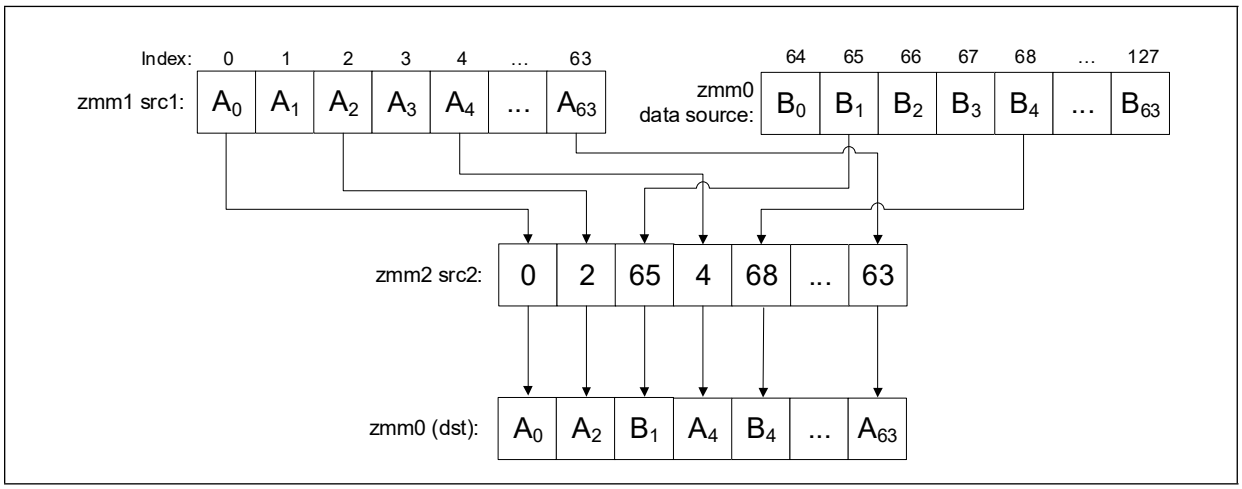


Figure 17-20. VPERMT2B Instruction Operation

VPERMT2B Operation:

```
// vpermt2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, * Src2;
data2= copy(Dst);
for(int i=0;i<64;i++){
        if(k1[i]){
            Dst[i]= Src2[i]>63 ? Src1[Src2 [i] & 63] : Dst[Src2[i] & 63]  ;
        }else{
            Dst[i]= zero_masking? 0 : Dst[i];
        }
}
```

The following example shows a 128-byte lookup table implementation.

C Code:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int numOfElements){
    for(int i = 0; i < numOfElements; i++) {
        out_bytes[i] = dictionary_bytes[in_bytes[i] & 127];
    }
}
```

Example 17-27. Improvement with VPERMI2B Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMI2B Implementation

//get data sent to function
mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
//Reorganize dictionary
vpmovzxbw zmm10, [rsi]
vpmovzxbw zmm15, [rsi+64]
vpsllw zmm15, zmm15, 8
vpord zmm10, zmm15, zmm10
vpmovzxbw zmm11, [rsi+32]
vpmovzxbw zmm15, [rsi+96]
vpsllw zmm15, zmm15, 8
vpord zmm11, zmm15, zmm11
//initialize constants
mov r10, 0x00400040
vpbroadcastw zmm12, r10d
mov r10, 0
vpbroadcastd zmm13, r10d
mov r10, 0x00ff00ff
vpbroadcastd zmm14, r10d
//start iterations
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpandd zmm2, zmm1, zmm12
vpcmpw k1, zmm2, zmm13, 4
vpermi2w zmm1, zmm10, zmm11
vpsrlw zmm1{k1}, zmm1, 8
vpandd zmm1, zmm1, zmm14
vpmovwb [rax+r8*1], zmm1
add r8, 32
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	vmovdqu32 zmm3, [rsi+64]
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermi2b zmm1, zmm2, zmm3
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

Base Measurement: 1x	Speedup: 5.3x

17.17.3 SELECT PACKED UNALIGNED BYTES FROM QUADWORD SOURCES (VPMULTISHIFTQB)

The VPMULTISHIFTQB instruction selects eight unaligned bytes from each input qword element of the second source operand and writes eight assembled bytes for each qword element in the destination operand.

The following figure shows a VPMULTISHIFTQB instruction operation example.

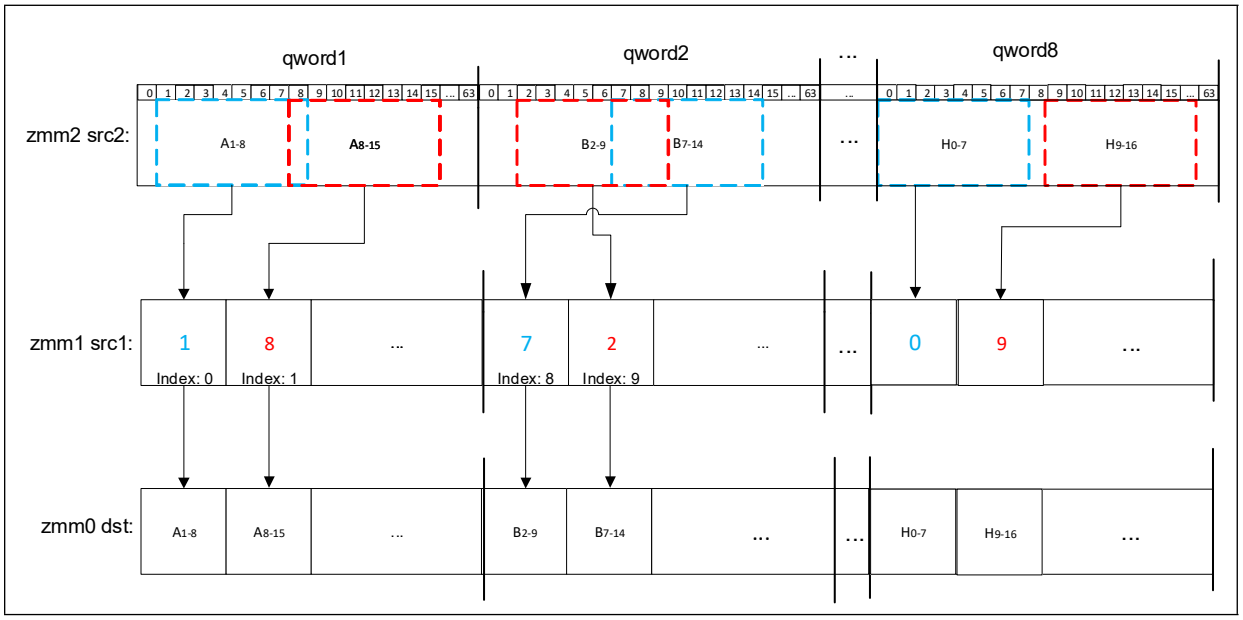


Figure 17-21. VPMULTISHIFTQB Instruction Operation

VPMULTISHIFTQB Operation:

```
// vpmultishiftqb Dst{k1},Src1,Src2
bool zero_masking=false;
unsigned char *Dst, * Src1;
unsigned __int64 *Src2;
bit * k1;
for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(k1[i*8 +j]){
                Dst[i*8 +j]= (src2[i]>> Src1[i*8 +j]) &0xFF  ;
            }else{
                Dst[i*8 +j]= zero_masking? 0 : Dst[i*8 +j];
            }
        }
}
```

The following example converts a 5-bit unsigned integer array to a 1-byte unsigned integer array.

C code:

```
void decompress (unsigned char* compressedData, unsigned char* decompressedData, int
numOfElements){
    for(int i = 0; i < numOfElements; i += 8){
        unsigned __int64 * data = (unsigned __int64 * )compressedData;
        decompressedData[i+0] = * data & 0x1f;
        decompressedData[i+1] = (*data >> 5 ) & 0x1f;
        decompressedData[i+2] = (*data >> 10 ) & 0x1f;
        decompressedData[i+3] = (*data >> 15 ) & 0x1f;
        decompressedData[i+4] = (*data >> 20 ) & 0x1f;
        decompressedData[i+5] = (*data >> 25 ) & 0x1f;
        decompressedData[i+6] = (*data >> 30 ) & 0x1f;
        decompressedData[i+7] = (*data >> 35 ) & 0x1f;
        compressedData += 5;
    }
}
```

Example 17-28. Improvement with VPMULTISHIFTQB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPMULTISHIFTQB Implementation

mov rdx, compressedData
mov r9, decompressedData
mov eax, numOfElements
shr eax,3
xor rsi, rsi
loop:
mov rcx, qword ptr [rdx]
mov r10, rcx
and r10, 0x1f
mov r11, rcx
mov byte ptr [r9+rsi*8], r10b
mov r10, rcx
shr r10, 0xa
add rdx, 0x5
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x2], r10b
mov r10, rcx
shr r10, 0xf
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x3], r10b
mov r10, rcx
shr r10, 0x14
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x4], r10b
mov r10, rcx
shr r10, 0x19
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x5], r10b
mov r10, rcx
shr r11, 0x5
shr r10, 0x1e
and r11, 0x1f
 	
shr rcx, 0x23
and r10, 0x1f
and rcx, 0x1f
mov byte ptr [r9+rsi*8+0x1], r11b
mov byte ptr [r9+rsi*8+0x6], r10b
mov byte ptr [r9+rsi*8+0x7], cl
inc rsi
cmp rsi, rax
jb loop
	//constants :
	__declspec (align(64)) const unsigned __int8
	permute_ctrl[64] = {
	0, 1, 2, 3, 4, 0, 0, 0
	5, 6, 7, 8, 9, 0, 0, 0
	10, 11, 12, 13, 14, 0, 0, 0
	15, 16, 17, 18, 19, 0, 0, 0
	20, 21, 22, 23, 24, 0, 0, 0
	25, 26, 27, 28, 29, 0, 0, 0
	30, 31, 32, 33, 34, 0, 0, 0
	35, 36, 37, 38, 39, 0, 0, 0
	};
	__declspec (align(64)) const unsigned __int8
	multishift_ctrl[64] = {
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	0, 5, 10, 15, 20, 25, 30, 35
	};
	//asm:
	mov rsi, compressedData
	mov rdi, decompressedData
	mov r8d, numOfElements
	lea r8, [rdi+r8]
	mov r9, 0x1F1F1F1F
	vpbroadcastd zmm12, r9d
	vmovdqu32 zmm10, permute_ctrl
	vmovdqu32 zmm11, multishift_ctrl
	
	loop:
	vmovdqu32 zmm1, [rsi]
	vpermb zmm2, zmm10, zmm1
	vpmultishiftqb zmm2, zmm11, zmm2
	vpandq zmm2, zmm12, zmm2
	vmovdqu32 [rdi], zmm2
	add rdi, 64
	add rsi, 40
	cmp rdi, r8
	jl loop

Base Measurement: 1x	Speedup: 26x

17.18 FMA LATENCY

When executing in 512-bit register port scheme, Port 0 FMA has a latency of four cycles, and Port 5 FMA has a latency of six cycles. Bypass can have a -2 (fast bypass) to +1 cycle delay. Therefore, instructions that execute on the Skylake microarchitecture FMA have a latency of four to seven cycles.

The instructions are divided into the following two groups.

- Group A Instructions: vadd*; vfmadd*; vfnmsub*; vfnmadd*; vfnmsub*; vmax*; vmin*; vmul*; vscalef*; vsub*; vcvt*; vgetexp*; vfixupimm*; vrange*; vgetmant*; vreduce*; vcmp*, vcomi*, vdpp*, vhadd*, vhsub*, vrndscale*, vround*
- Group B Instructions: vpmaddubsw; vpmaddwd; vpmuldq; vpmulhrsw; vpmulhuw; vpmulhw; vpmullw; vpmuludq

The FMA unit supports fast bypass when all instruction sources come from the FMA unit. In this case Group A has a latency of four cycles for both ports 0 and 5, and Group B has a latency of five cycles for both ports 0 and 5.

The figure below explains fast bypass when all sources come from the FMA unit.

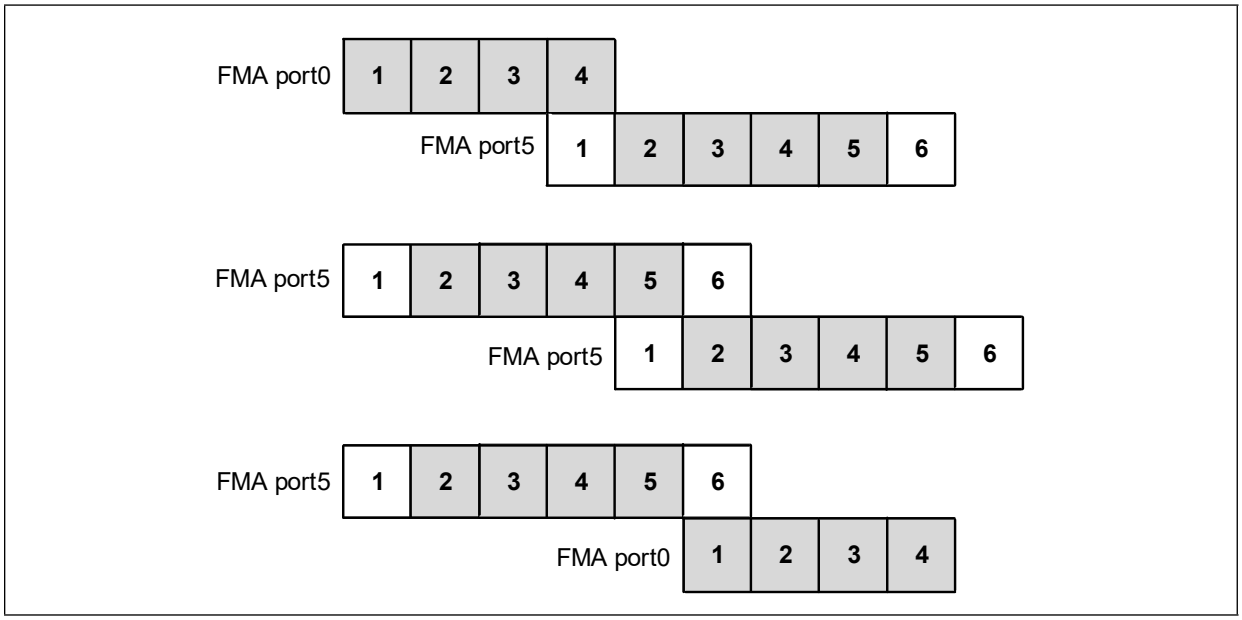


Figure 17-22. Fast Bypass When All Sources Come from FMA Unit

The gray boxes represent compute cycles. The white boxes represent data transfer for the port5 FMA unit.

If fast bypass is not used, that is, when not all sources come from the FMA unit, group A instructions have a latency of four cycles on Port0 and six cycles on port5, while group B instructions have an additional cycle and hence have a latency of five cycles on Port0 and seven cycles on port5.

The following table summarizes the FMA unit latency for the various options.

Table 17-9. FMA Unit Latency

	Fast Bypass (FMA Data Reuse)		No Fast Bypass (No FMA Data Reuse)	
Instruction Group	**Port 0**	**Port 5**	**Port 0**	**Port 5**
Group A	4	4	4	6

Table 17-9. FMA Unit Latency (Contd.)

	Fast Bypass (FMA Data Reuse)		No Fast Bypass (No FMA Data Reuse)	
Instruction Group	**Port 0**	**Port 5**	**Port 0**	**Port 5**
Group B	5	5	5	7

17.19 MIXING INTEL® AVX OR INTEL® AVX-512 EXTENSIONS WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) CODE

There are two main instruction groups that affect the processor states:

- Group A: Instruction types that either set bits 128-511 of vector registers 0-15 to zero, or do not modify them at all.
  - Intel SSE instructions.
  - 128-bit Intel AVX instructions, 128-bit Intel AVX-512 instructions.
  - 256-bit (ymm16-ymm31) Intel AVX-512 instructions.
  - 512-bit (zmm16-zmm31) Intel AVX-512 instructions.
  - AVX-512 instructions that write to mask registers k0-k7.
  - GPR instructions.
- Group B: Instructions types that modify bits 128-511 of vector registers 0-15.
  - 256-bit (ymm0-ymm15) Intel AVX instructions, Intel AVX-512 instructions.
  - 512-bit (zmm0-zmm15) Intel AVX-512 instructions.

The following figure illustrates Skylake Server microarchitecture's model for mixing Intel AVX instructions or Intel AVX-512 instructions with Intel SSE instructions.

The implementation is similar to Skylake client microarchitecture, where every Intel SSE instruction executed in Dirty Upper State (2) needs to preserve bits 128-511 of the destination register, and therefore the operation has an additional dependency on the destination register and a blend operation with bits 128-511.

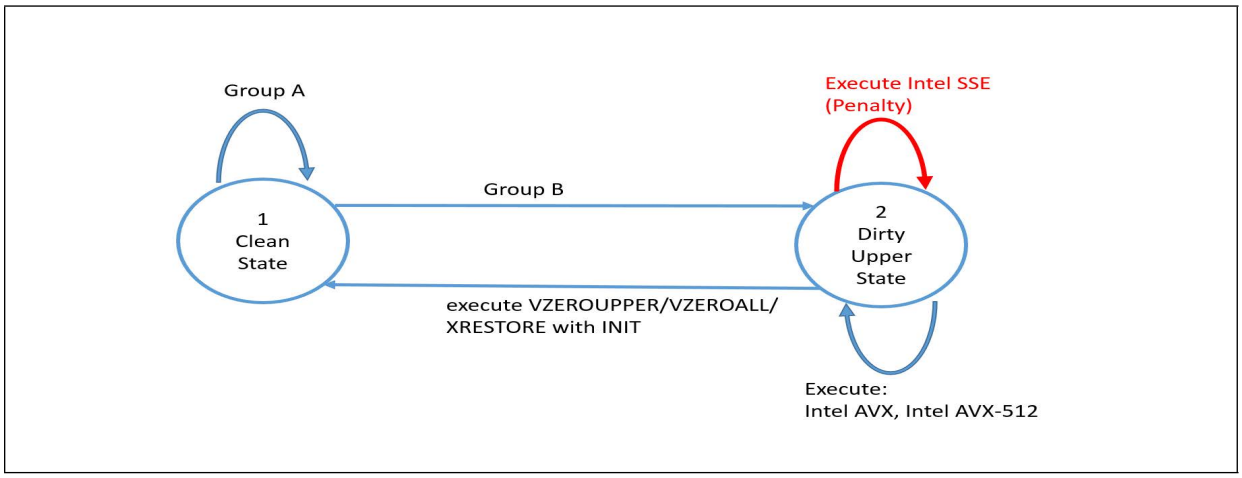


Figure 17-23. Mixing Intel AVX Instructions or Intel AVX-512 Instructions with Intel SSE Instructions

Recommendations:

- When mixing group B instructions with Intel SSE instructions, or suspecting that such a mixture might occur, use the VZEROUPPER instruction whenever a transition is expected.
- Add VZEROUPPER after group B instructions were executed and before any function call that might lead to an Intel SSE instruction execution.
- Add VZEROUPPER at the end of any function that uses group B instructions.
- Add VZEROUPPER before thread creation if not already in a clean state so that the thread does not inherit a Dirty Upper State.

17.20 MIXING ZMM VECTOR CODE WITH XMM/YMM

Skylake microarchitecture has two port schemes, one for using 256-bit or less registers, and another for using 512-bit registers.

When using registers up to or including 256 bits, FMA operations dispatch to ports 0 and 1 and SIMD operations dispatch to ports 0, 1 and 5. When using 512-bit register operations, both FMA and SIMD operations dispatch to ports 0 and 5.

The maximum register width in the reservation station (RS) determines the 256 or 512 port scheme.

Notice that when using AVX-512 encoded instructions with YMM registers, the instructions are considered to be 256-bit wide.

The result of the 512-bit port scheme is that XMM or YMM code dispatches to two ports (0 and 5) instead of three ports (0, 1, and 5) and may have lower throughput and longer latency compared to the 256-bit port scheme.

Example 17-29. 256-bit Code vs. 256-bit Code Mixed with 512-bit Code

256-bit Code Only	256-bit Code Mixed with 512-bit Code
Loop: vpbroadcastd ymm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnle Loop	Loop: vpbroadcastd zmm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnle Loop
Baseline 1x	Slowdown: 1.3x

In the 256-bit code only example, the FMAs are dispatched to ports 0 and 1, and *permd* is dispatched to port 5 as the broadcast instruction is 256 bits wide. In the 256-bit and 512-bit mixed code example, the broadcast is 512 bits wide; therefore, the processor uses the 512-bit port scheme where the FMAs dispatch to ports 0 and 5 and *permd* to port 5, thus increasing the pressure on port 5.

17.21 SERVERS WITH A SINGLE FMA UNIT

Some processors based on Skylake microarchitecture have two Intel AVX-512 FMA units, on ports 0 and 5, while other processors based on Skylake microarchitecture have a single Intel AVX-512 FMA unit, which is located on port 0.

Code that is optimized to run on a processor with two FMA units might not be optimal when run on a processor with one FMA unit.

The following example code shows how to detect whether a system has one or two Intel AVX-512 FMA units. It includes the following:

- An Intel AVX-512 warmup.
- A function that executes only FMA instructions.
- A function that executes both FMA and shuffle instructions.
- Code that, based on the results of these two tests, identifies whether the processor has one or two FMA units.

Notice that each test is executed three times to improve test accuracy.

To reduce the program overhead, it is highly recommended not to execute this test in every function call, but as part of installation, or once at startup.

The differentiation between the two processors is based on the ratio between the two throughput tests. Processors with two FMA units are able to run the FMA-only test twice as fast as the FMA and shuffle test. However, a processor with one FMA unit will run both tests at the same speed.

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
#include <string.h>
#include <stdlib.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdint.h>

static uint64_t rdtsc(void) {
  unsigned int ax, dx;

  __asm__ __volatile__ ("rdtsc" : "=a"(ax), "=d"(dx));

  return ((((uint64_t)dx) << 32) | ax);
}

uint64_t fma_shuffle_tpt(uint64_t loop_cnt){
   uint64_t loops =  loop_cnt;
   __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
   __declspec(align(64)) int shuf_vec[16] = {0, 1, 2, 3,4, 5, 6, 7,8, 9, 10, 11,12, 13, 14, 15};
    __asm
       {
       vmovups zmm0, [one_vec]
       vmovups zmm1, [one_vec]
       vmovups zmm2, [one_vec]
       vmovups zmm3, [one_vec]
       vmovups zmm4, [one_vec]
       vmovups zmm5, [one_vec]
       vmovups zmm6, [one_vec]
       vmovups zmm7, [one_vec]
       vmovups zmm8, [one_vec]
       vmovups zmm9, [one_vec]
vmovups zmm10, [one_vec]
       vmovups zmm11, [one_vec]
       vmovups zmm12, [shuf_vec]
       vmovups zmm13, [shuf_vec]
       vmovups zmm14, [shuf_vec]
       vmovups zmm15, [shuf_vec]
       vmovups zmm16, [shuf_vec]
       vmovups zmm17, [shuf_vec]
       vmovups zmm18, [shuf_vec]
       vmovups zmm19, [shuf_vec]
       vmovups zmm20, [shuf_vec]
       vmovups zmm21, [shuf_vec]
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
        vmovups zmm22, [shuf_vec]
        vmovups zmm23, [shuf_vec]
        vmovups zmm30, [shuf_vec]
        mov rdx, loops
loop1:
        vfmadd231pd zmm0, zmm0, zmm0
        vfmadd231pd zmm1, zmm1, zmm1
        vfmadd231pd zmm2, zmm2, zmm2
        vfmadd231pd zmm3, zmm3, zmm3
        vfmadd231pd zmm4, zmm4, zmm4
        vfmadd231pd zmm5, zmm5, zmm5
        vfmadd231pd zmm6, zmm6, zmm6
        vfmadd231pd zmm7, zmm7, zmm7
        vfmadd231pd zmm8, zmm8, zmm8
        vfmadd231pd zmm9, zmm9, zmm9
        vfmadd231pd zmm10, zmm10, zmm10
        vfmadd231pd zmm11, zmm11, zmm11
        vpermd zmm12, zmm30, zmm30
        vpermd zmm13, zmm30, zmm30
        vpermd zmm14, zmm30, zmm30
        vpermd zmm15, zmm30, zmm30
        vpermd zmm16, zmm30, zmm30
        vpermd zmm17, zmm30, zmm30
        vpermd zmm18, zmm30, zmm30
        vpermd zmm19, zmm30, zmm30
        vpermd zmm20, zmm30, zmm30
        vpermd zmm21, zmm30, zmm30
        vpermd zmm22, zmm30, zmm30
        vpermd zmm23, zmm30, zmm30
        dec rdx
        jg loop1
        }
}
uint64_t fma_only_tpt(int loop_cnt){
  uint64_t loops =  loop_cnt;
  __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
        __asm
        {
        vmovups zmm0, [one_vec]
        vmovups zmm1, [one_vec]
        vmovups zmm2, [one_vec]
        vmovups zmm3, [one_vec]
        vmovups zmm4, [one_vec]
        vmovups zmm5, [one_vec]
        vmovups zmm6, [one_vec]
        vmovups zmm7, [one_vec]
        vmovups zmm8, [one_vec]
        vmovups zmm9, [one_vec]
        vmovups zmm10, [one_vec]
        vmovups zmm11, [one_vec]
        mov rdx, loops
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
   loop1:
      vfmadd231pd zmm0, zmm0, zmm0
      vfmadd231pd zmm1, zmm1, zmm1
      vfmadd231pd zmm2, zmm2, zmm2
      vfmadd231pd zmm3, zmm3, zmm3
vfmadd231pd zmm4, zmm4, zmm4
      vfmadd231pd zmm5, zmm5, zmm5
      vfmadd231pd zmm6, zmm6, zmm6
      vfmadd231pd zmm7, zmm7, zmm7
      vfmadd231pd zmm8, zmm8, zmm8
      vfmadd231pd zmm9, zmm9, zmm9
      vfmadd231pd zmm10, zmm10, zmm10
      vfmadd231pd zmm11, zmm11, zmm11
      dec rdx
      jg loop1
      }
}

int main()
{
  int i;
  uint64_t fma_shuf_tpt_test[3];
  uint64_t fma_shuf_tpt_test_min;
  uint64_t fma_only_tpt_test[3];
  uint64_t fma_only_tpt_test_min;
  uint64_t start = 0;
  uint64_t number_of_fma_units_per_core = 2;

  /*********************************************************/
  /* Step 1: Warmup */
  /*********************************************************/
  fma_only_tpt(100000);

  /*********************************************************/
  /* Step 2: Execute FMA and Shuffle TPT Test */
  /*********************************************************/

  for(i = 0; i < 3; i++){
     start = rdtsc();
     fma_shuffle_tpt(1000);
     fma_shuf_tpt_test[i] = rdtsc() - start;
  }

  /*********************************************************/
  /* Step 3: Execute FMA only TPT Test  */
  /*********************************************************/
  for(i = 0; i < 3; i++){
     start = rdtsc();
     fma_only_tpt(1000);
     fma_only_tpt_test[i] = rdtsc() - start;
  }
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
    /*******************************************************/
    /* Step 4: Decide if 1 FMA server or 2 FMA server */
    /*******************************************************/
    fma_shuf_tpt_test_min = fma_shuf_tpt_test[0];
    fma_only_tpt_test_min = fma_only_tpt_test[0];
    for(i = 1; i < 3; i++){
        if ((int)fma_shuf_tpt_test[i] < (int)fma_shuf_tpt_test_min) fma_shuf_tpt_test_min = fma_shuf_tpt_test[i];
        if ((int)fma_only_tpt_test[i] < (int)fma_only_tpt_test_min) fma_only_tpt_test_min = fma_only_tpt_test[i];
    }

    if(((double)fma_shuf_tpt_test_min/(double)fma_only_tpt_test_min) < 1.5){
        number_of_fma_units_per_core = 1;
    }

    printf("%d FMA server\n", number_of_fma_units_per_core);
    return 0;
}
```

17.22 GATHER/SCATTER TO SHUFFLE (G2S/STS)

17.22.1 GATHER TO SHUFFLE IN STRIDED LOADS

In cases where there is data locality between gathered elements in memory, performance can be improved by replacing the gather instruction with a software sequence.

This section discusses the very common strided load pattern. Strided loads are sets of loads where the offset in memory between two consecutive loads is constant.

The following examples show three different code variations performing an Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

Consider the following C code:

```
    for(int i=0;i<len;i++){
            Real_buffer[i] = Complex_buffer[i].real;
            Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

Example 17-31. Gather to Shuffle in Strided Loads Example

Alternative 1: Intel® AVX-512 vpgatherdd	Alternative 2: G2S Using Intel® AVX-512 vpermi2d
loop: vpcmpeqb k1, xmm0, xmm0 vpcmpeqb k2, xmm0, xmm0 movsxd rdx, edx movsxd rdi, esi inc esi	vmovups zmm4, [rdx+r9*8] vmovups zmm0, [rdx+r9*8+0x40] vmovups zmm5, [rdx+r9*8+0x80] vmovups zmm1, [rdx+r9*8+0xc0] vmovaps zmm2, zmm7 vmovaps zmm3, zmm7

Example 17-31. Gather to Shuffle in Strided Loads Example (Contd.)

Alternative 1: Intel® AVX-512 vpgatherdd	Alternative 2: G2S Using Intel® AVX-512 vpermi2d

shl rdi, 0x7
vpxord zmm2, zmm2, zmm2
lea rax, [r8+rdx*8]
add edx, 0x20
vpgatherdd zmm2{k1}, [rax+zmm1*4]
vpxord zmm3, zmm3, zmm3
vpxord zmm4, zmm4, zmm4
vpxord zmm5, zmm5, zmm5
vpgatherdd zmm3{k2}, [rax+zmm0*4]
vpcmpeqb k3, xmm0, xmm0
vpcmpeqb k4, xmm0, xmm0
vmovups [r9+rdi*1], zmm2
vmovups [rcx+rdi*1], zmm3
vpgatherdd zmm4{k3}, [rax+zmm1*4+0x80]
vpgatherdd zmm5{k4}, [rax+zmm0*4+0x80]
vmovups [r9+rdi*1+0x40], zmm4
vmovups [rcx+rdi*1+0x40], zmm5
cmp esi, r14d
jb loop
	vpermi2d zmm2, zmm4, zmm0
	vpermt2d zmm4, zmm6, zmm0
	vpermi2d zmm3, zmm5, zmm1
	vpermt2d zmm5, zmm6, zmm1
	vmovdqu32 [rcx+r9*4], zmm2vmovdqu32
	[rcx+r9*4+0x40], zmm3
	vmovdqu32 [r8+r9*4], zmm4
	vmovdqu32 [r8+r9*4+0x40], zmm5add r9, 0x20
	cmp r9, r10
	jb loop

Baseline 1x	Speedup: 4.8x

The following constants were loaded into zmm registers and used as gather and permute indices:

Zmm0 (Alternative 1), zmm6 (Alternative 2)

__declspec (align(64)) const __int32 gather_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

Zmm1 (Alternative 1), zmm7 (Alternative 2)

__declspec (align(64)) const __int32 gather_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Recommendation: For best performance, replace strided loads where the stride is short, with a sequence of loads and permutes.

17.22.2 SCATTER TO SHUFFLE IN STRIDED STORES

The following is an Scatter to Shuffle example that replaces scatter with permute and store instructions

Consider the following C code:

```
for(int i=0;i<len;i++){
        Complex_buffer[i].real = Real_buffer[i];
        Complex_buffer[i].imag = Imaginary_buffer[i];}
```

Example 17-32. Gather to Shuffle in Strided Stores Example

Alternative 1: Intel® AVX-512 vscatterdps	Alternative 2: S2S using Intel® AVX-512 vpermi2d

loop:
vpcmpeqb k1, xmm0, xmm0
lea r11, [r8+rcx*4]
vpcmpeqb k2, xmm0, xmm0
vmovups zmm2, [rax+rsi*4]
vmovups zmm3, [r9+rsi*4]
vscatterdps [r11+zmm1*4]{k1}, zmm2
vscatterdps [r11+zmm0*4]{k2}, zmm3
add rsi, 0x10
add rcx, 0x20
cmp rsi, r10
jl loop
	loop:
	vmovups zmm4, [rax+r8*4]
	vmovups zmm2, [r10+r8*4]
	vmovaps zmm3, zmm1
	add r8, 0x10
	vpermi2d zmm3, zmm4, zmm2
	vpermt2d zmm4, zmm0, zmm2
	vmovups [r9+rsi*4], zmm3
	vmovups [r9+rsi*4+0x40], zmm4
	add rsi, 0x20
	cmp r8, r11
	jl loop

Baseline 1x	Speedup: 4.4x

The following constants were used as scatter indices:

Zmm1:

```
__declspec (align(64)) const __int32 scatter_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};
```

Zmm0:

```
__declspec (align(64)) const __int32 scatter_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};
```

The following constants were used as permute indices:

Zmm1:

```
__declspec (align(64)) const __int32 first_half[16] = {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23};
```

Zmm0:

```
__declspec (align(64)) const __int32 second_half[16] = {8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31};
```

17.22.3 GATHER TO SHUFFLE IN ADJACENT LOADS

In cases where the gathered elements are grouped into adjacent sequences, the gather instruction can be replaced by a software sequence to improve performance.

The following example shows how to load vectors when elements are adjacent.

Notice that in this case the order of the elements in the arrays is set according to an index buffer and therefore the software optimization discussed in Section 17.22.1 is not applicable in this case.

Consider the following C code:

```
typedef struct{
    double var[4];
} ElemStruct;

const int* indices = Indices;
const ElemStruct *in = (const ElemStruct*) InputBuffer;
double* restrict out = OutputBuffer;

for (int i = 0; i < width; i++){
        for (int j = 0; j < 4; j++){
            out[i*4+j] = in[indices[i]].var[j];
    }
}
```

Example 17-33. Gather to Shuffle in Adjacent Loads Example

Alternative 1: vgatherdpd Implementation	Alternative 2: Load and Masked broadcast

loop:
vpbroadcastd ymm3, [r9+rsi*4]
mov r15d, esi
vpbroadcastd xmm2, [r9+rsi*4+0x4]
add rsi, 0x2
vpbroadcastd ymm3{k1}, xmm2
vpmulld ymm4, ymm3, ymm1
vpaddd ymm5, ymm4, ymm0
vpcmpeqb k2, xmm0, xmm0
shl r15d, 0x2
movsxd r15, r15d
vpxord zmm6, zmm6, zmm6
vgatherdpd zmm6{k2}, [r10+ymm5*1]
vmovups [r11+r15*8], zmm6
cmp rsi, rdi
jl loop
	loop:
	movsxd r11, [r10+rcx*4]
	shl r11, 0x5
	vmovupd ymm0, [r9+r11*1]
	movsxd r11, [r10+rcx*4+0x4]
	shl r11, 0x5
	vbroadcastf64x4 zmm0{k1}, [r9+r11*1]
	mov r11d, ecx
	shl r11d, 0x2
	add rcx, 0x2
	movsxd r11, r11d
	vmovups [r8+r11*8], zmm0
	cmp rcx, rsi
	jl loop

Baseline 1x	Speedup: 2.2x

The following constants were used in the vgatherdpd implementation:

ymm0:

```
__declspec (align(64)) const __int32 index_inc[8] = {0, 8, 16, 24, 0, 8, 16, 24};
```

ymm1:

```
__declspec (align(64)) const __int32 index_scale[8] = {32, 32, 32, 32, 32, 32, 32, 32};
```

K1 register value is 0xF0.

17.23 DATA ALIGNMENT

This section explains the benefit of aligning data when using the Intel AVX-512 instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

The C code below is a C implementation of SAXPY.

```
for (int i = 0; i < n; i++)
{
c[i] = alpha * a[i] + b[i];
}
```

17.23.1 ALIGN DATA TO 64 BYTES

Aligning data to vector length is recommended. For best results, when using Intel AVX-512 instructions, align data to 64 bytes.

When doing a 64-byte Intel AVX-512 unaligned load/store, every load/store is a cache-line split, since the cache-line is 64 bytes. This is double the cache line split rate of Intel AVX2 code that uses 32-byte registers. A high cache-line split rate in memory-intensive code can cause poor performance.

The following table shows how the performance of the memory intensive SAXPY code is affected by misaligning input and output buffers. The data in the table is based on the following code.

Example 17-34. Data Alignment

```
__asm {
        mov rax, src1
        mov rbx, src2
        mov rcx, dst
        mov rdx, len
        xor rdi, rdi
        vbroadcastss zmm0, alpha
mainloop:
        vmovups zmm1, [rax]
        vfmadd213ps zmm1, zmm0, [rbx]
        vmovups [rcx], zmm1

        vmovups zmm1, [rax+0x40]
        vfmadd213ps zmm1, zmm0, [rbx+0x40]
        vmovups [rcx+0x40], zmm1

        vmovups zmm1, [rax+0x80]
        vfmadd213ps zmm1, zmm0, [rbx+0x80]
        vmovups [rcx+0x80], zmm1
```

Example 17-34. Data Alignment (Contd.)

```
    vmovups zmm1, [rax+0xC0]
        vfmadd213ps zmm1, zmm0, [rbx+0xC0]
        vmovups [rcx+0xC0], zmm1

        add rax, 256
        add rbx, 256
        add rcx, 256
        add rdi, 64
        cmp rdi, rdx
        jl mainloop
}
```

The Table 17-10 summarizes the data alignment effects on SAXPY performance with speedup values for the various options.

Table 17-10. Data Alignment Effects on SAXPY Performance vs. Speedup Value

Data Alignment Effects on SAXPY Performance	Speedup
Alternative 1: Both sources and the destination are 64-byte aligned.	Baseline, 1.0
Alternative 2: Both sources are 64-byte aligned, destination has a 4 byte offset from the alignment.	0.66x
Alternative 3: Both sources and the destinations have 4 bytes offset from the alignment.	0.59x
Alternative 4: One source has a 4 byte offset from the alignment, the other source and the destination are 64-byte aligned.	0.77x

17.24 DYNAMIC MEMORY ALLOCATION AND MEMORY ALIGNMENT

Consider the following structure:

```
float3_SOA {
 __declspec(align(64)) float x[16];
 __declspec(align(64)) float y[16];
 };
```

The memory allocated for the structure is aligned to 64 bytes if you use this structure as follows:

```
float3_SOA f;
```

However, if you use dynamic memory allocation as follows, the declspec directive is ignored and the 64-byte memory alignment is not guaranteed:

```
float3_SOA* stPtr = new float3_SOA();
```

In this case, you should use dynamic aligned memory allocation and/or redefine operator *new*.

Recommendation: Align data to 64 bytes, when possible, using the following guidelines.

- Use dynamic data alignment using the _mm_malloc intrinsic instruction with the Intel® Compiler, or _aligned_malloc of the Microsoft* Compiler.

— For example:

//dynamically allocating 64byte aligned buffer with 2048 float elements.

InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 64);

- Use static data alignment using __declspec(align(64)).
  — For example:

//Statically allocating 64byte aligned buffer with 2048 float elements.

__declspec(align(64)) float InputBuffer[2048];

17.25 DIVISION AND SQUARE ROOT OPERATIONS

It is possible to speed up single-precision divide and square root calculations using the VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions. These instructions yield an approximation (with 14 bits accuracy) of the Reciprocal Square Roots / Reciprocal Divide of their input.

The Intel AVX-512 implementation of these instructions is pipelined and has:

- For 256-bit vectors: latency of four cycles with a throughput of one instruction every cycle.
- For 512-bit vectors: latency of six cycles with a throughput of one instruction every two cycles.

Skylake microarchitecture introduces the packed-double (PD) variants of reciprocal square-root and reciprocal divide: VRSQRT14PD and VRCP14PD (respectively).

The VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions can be used with a single Newton-Raphson iteration or other polynomial approximation to achieve almost the same precision as the VDIVPS and VSQRTPS instructions (see the Intel® 64 and IA-32 Architectures Software Developer's Manual for more information on these instructions), and may yield a much higher throughput.

If the full precision (IEEE) must be maintained, a low latency and high throughput can be achieved due to the significant performance improvement of the Skylake microarchitecture to DIVPS and SQRTPS, comparing to their performance on previous microarchitectures. This is illustrated in Figure 17-13.

NOTE

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

The following sections show the operations with recommended calculation methods depending on the desired accuracy level.

NOTE

There are two definitions for approximation error of a value and it's approximation ν_{approx}:

Absolute error = $|\nu - \nu_{approx}|$

Relative error = $|\nu - \nu_{approx}| / |\nu|$

In this chapter, the "number of bits" error is relative, and not the error of absolute values.

The value ν to which we compare our approximation should be as accurate as possible, better double accuracy.

17.25.1 DIVIDE AND SQUARE ROOT APPROXIMATION METHODS

Table 17-11. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Single Precision)

Operation	Accuracy	Recommended Method
Divide	24 bits (IEEE)	DIVPS
	23 bits	RCP14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RCP14PS + MULPS
Reciprocal Square Root	22 bits	SQRTPS + DIVPS
	23 bits	RSQRT14PS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS
Square Root	24 bits (IEEE)	SQRTPS
	23 bits	RSQRT14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS + MULPS

Table 17-12. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Double Precision)

Operation	Accuracy	Recommended Method
Divide	53 bits (IEEE)	DIVPD
	52 bits	RCP14PD + MULPD + 2 Newton-Raphson iterations
	26 bits	RCP14PD + MULPD + 1 Newton-Raphson iterations
	14 bits	RCP14PD + MULPD
Reciprocal Square Root	53 bits (IEEE)	SQRTPD + DIVPD
	52 bits	RSQRT14PD+2 N-R + error correction or SQRTPD + DIVPD
	50 bits	RSQRT14PD + Polynomial approximation
	26 bits	RSQRT14PD+1 N-R
	14 bits	RSQRT14PD
Square Root	51 bits (IEEE)	SQRTPD
	52 bits	RSQRT14PD + MULPD + Polynomial approximation
	26 bits	RSQRT14PD + MULPD + 1 N-R
	14 bits	RSQRT14PD + MULPD

17.25.2 DIVIDE AND SQUARE ROOT PERFORMANCE

Performance of vector divide and square root operations on Broadwell and Skylake microarchitectures is shown

below.

Table 17-13. 256-bit Intel AVX2 Divide and Square Root Instruction Performance

Broadwell Microarchitecture	DIVPS	SQRTPS	DIVPD	SQRTPD
Latency	17	21	23	35
Throughput	10	14	16	28
Skylake Microarchitecture	DIVPS	SQRTPS	DIVPD	SQRTPD
Latency	11	12	14	18
Throughput	5	6	8	12

Table 17-14. 512-bit Intel AVX-512 Divide and Square Root Instruction Performance

Skylake Microarchitecture	DIVPS	SQRTPS	DIVPD	SQRTPD
Latency	17	19	23	31
Throughput	10	12	16	24

17.25.3 APPROXIMATION LATENCIES

This section shows the latency and throughput for the approximation methods, and DIV and SQRT instructions. The tables below show that in most cases the throughput gain of the approximation methods is (at least) double that of their IEEE counterparts, in simple loops that compute division or square root.

The throughput benefits of approximation sequences are diminished when the loop iterations contain a lot of other computation (besides divide or square root).

As a rule of thumb, approximations of near-IEEE accuracy are recommended when the loop iteration contains no more than eight to ten additional single precision operations, or no more than twelve to fifteen additional double precision operations. The tables below show that these accurate approximations are beneficial for throughput optimizations only. The less accurate approximations can help with latency, as well as throughput.

It should also be mentioned that Newton-Raphson approximations do not handle the following special cases correctly: denormal inputs, zeros, or Infinities. Some sequences also lose accuracy for nearly denormal inputs, due to underflow in intermediate steps. While zero and Infinity inputs are relatively easy to fix with a few additional operations (as done in some of the sequences below), denormal divisors cannot be addressed without significant performance impact. The approximation sequences work best for “middle-of-the-range” inputs that are not close to overflow or underflow thresholds.

The table below shows the latency and throughput of single precision Intel AVX-512 divide and square root instructions, compared to the approximation methods on Skylake microarchitecture.

Table 17-15. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Single Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPS	24 bits (IEEE)	5	11	10	17
	RCP14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	2	16	3	20
	RCP14PS + MULPS	14 bits	1	8	2	10-12
Square root	SQRTPS	24 bits (IEEE)	6	12	12	19
	RSQRT14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	3	16	5	20
	RSQRT14PS + MULPS	14 bits	2	9	3	12
Reciprocal square root	SQRTPS + DIVPS	22 bits	11	23	22	36
	RSQRT14PS + 1 Newton-Raphson Iteration	23 bits	3.67	20	4.89	25
	RSQRT14PS	14 bits	1	4	2	6

Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPD	53 bits (IEEE)	8	14	16	23
	RCP14PD + MULPD + 2 Newton-Raphson Iteration	22 bits	3.2	27	4.7	28.4
	RCP14PD + MULPD + 1 Newton-Raphson Iteration	26 bits	2	16	3	20
	RCP14PD + MULPD	14 bits	1	8	2	10-12

Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision (Contd.)

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Square root	SQRTPD	53 bits (IEEE)	12	18	24	31
	RSQRT14PD + MULPD + Polynomial Approximation	22 bits	4.82	24.54[1]	6.4	28.48[1]
	RSQRT14PD + MULPD + 1 N-R	26 bits	3.76	17	5	20
	RSQRT14PD + MULPD	14 bits	2	9	3	12
Reciprocal square root	SQRTPD + DIVPD	51 bits	20	32	40	53
	RSQRT14PD + 2-NR + ErrorCcorrection	52 bits	5	29.38	6.53	34
	RSQRT14PD+2 N-R	50 bits	3.79	25.73	5.51	30
	RSQRT14PD+1 N-R	26 bits	2.7	18	4.5	21.67
	RSQRT14PD	14 bits	1	4	2	6

NOTES:

1. These numbers are not rounded because their code sequence contains several FMA (Fused-multiply-add) instructions, which have a varying latency of 4/6. Therefore the latency for these sequences is not necessarily fixed.

17.25.4 CODE SNIPPETS

Example 17-35. Vectorized 32-bit Float Division, 24 Bits

Single Precision, Divide, 24 Bits (IEEE)

float a = 10;
float b = 5;
 	
__asm {
  vbroadcastss zmm0, a// fill zmm0 with 16 elements of a
  vbroadcastss zmm1, b// fill zmm1 with 16 elements of b
  vdivps zmm2, zmm0, zmm1// zmm2 = 16 elements of a/b
}

Example 17-36. Vectorized 32-bit Float Division, 23 and 14 Bits

Single Precision, Divide, 23 Bits	Single Precision, Divide, 14 Bits

/* Input:
zmm0 = vector of a's
zmm1 = vector of b's
Output:
zmm3 = vector of a/b
 	
*/
 	
__asm {
vrcp14ps zmm2, zmm1
vmulps zmm3, zmm0, zmm2
vmovaps zmm4, zmm0
vfnmadd231ps zmm4, zmm3, zmm1
vfmadd231ps zmm3, zmm4, zmm2
}
	/* Input:
	zmm0 = vector of a's
	zmm1 = vector of b's
	Output:
	zmm2 = vector of a/b
	
	*/
	
	__asm {
	vrcp14ps zmm2, zmm1
	vmulps zmm2, zmm0, zmm2
	}

Example 17-37. Reciprocal Square Root, 22 Bits

Single Precision, Reciprocal Square Root, 22 Bits

/* Input:
zmm0 = vector of a's
zmm1 = vector of 1's
Output:
zmm2 = vector of 1/sqrt (a)
*/
 	
float one = 1.0;
 	
__asm {
vbroadcastss zmm1, one// zmm1 = vector of 16 1's
vsqrtps zmm2, zmm0
vdivps zmm2, zmm1, zmm2
}

Example 17-38. Reciprocal Square Root, 23 and 14 Bits

Single Precision, Reciprocal Square Root, 23 Bits	Single Precision, Reciprocal Square Root, 14 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm2 = vector of 1/sqrt (a)
*/
 	
float half = 0.5;
 	
__asm {
vbroadcastss zmm1, half// zmm1 = vector of 16 0.5's
vrsqrt14ps zmm2, zmm0
vmulps zmm3, zmm0, zmm2
vmulps zmm4, zmm1, zmm2
vfnmadd231ps zmm1, zmm3, zmm4
vfmsub231ps zmm3, zmm0, zmm2
 	
vfnmadd231ps zmm1, zmm4, zmm3
vfmadd231ps zmm2, zmm2, zmm1
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm2 = vector of 1/sqrt (a)
	*/
	
	__asm {
	vrsqrt14ps zmm2, zmm0
	}

Example 17-39. Square Root , 24 Bits

Single Precision, Square Root, 24 Bits (IEEE)

/* Input:
zmm0 = vector of a's
Output:
zmm2 = vector of sqrt (a)
*/
 	
__asm {
vsqrtps zmm2, zmm0
}

Example 17-40. Square Root , 23 and 14 Bits

Single Precision, Square Root, 23 Bits	Single Precision, Square Root, 14 Bits
/* Input: zmm0 = vector of a's Output: zmm0 = vector of sqrt (a) */ float half = 0.5;	/* Input: zmm0 = vector of a's Output: zmm0 = vector of sqrt (a) */

Example 17-40. (Contd.)Square Root (Contd.), 23 and 14 Bits

Single Precision, Square Root, 23 Bits	Single Precision, Square Root, 14 Bits

__asm {
vbroadcastss zmm3, half
vrsqrt14ps zmm1, zmm0
vfpclassps k2, zmm0, 0xe
vmulps zmm2, zmm0, zmm1, {rn-sae}
vmulps zmm1, zmm1, zmm3
knotw k3, k2
vfnmadd231ps zmm0{k3}, zmm2, zmm2
vfmadd213ps zmm0{k3}, zmm1, zmm2
}
	__asm {
	vrsqrt14ps zmm1, zmm0
	vfpclassps k2, zmm0, 0xe
	knotw k3, k2
	vmulps zmm0{k3}, zmm0, zmm1
	}

Example 17-41. Dividing Packed Doubles, 53 and 52 Bits

Double Precision, Divide, 53 Bits (IEEE)	Double Precision, Divide, 52 Bits

/* Input:
zmm0 = vector of a's
zmm1 = vector of b's
Output:
zmm2 = vector of a/b
*/
 	
__asm {
vdivpd zmm2, zmm0, zmm1
}
	/* Input:
	zmm15 = vector of a's
	zmm0 = vector of b's
	Output:
	zmm0 = vector of a/b
	*/
	
	double One = 1.0;
	
	__asm {
	vrcp14pd zmm1, zmm0
	vmovapd zmm4, zmm0
	vbroadcastsd zmm2, one
	vfnmadd213pd zmm0, zmm1, zmm2, {rn-sae}
	vfpclasspd k2, zmm1, 0x1e
	vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}}
	knotw k3, k2
	vfnmadd213pd zmm4, zmm0, zmm2, {rn-sae}
	vblendmpd zmm0 {k2}, zmm0, zmm1
	vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-sae}
	vmulpd zmm0, zmm0, zmm15
	}

Example 17-42. Dividing Packed Doubles, 26 and 14 Bits

Double Precision, Divide, 26 Bits	Double Precision, Divide, 14 Bits
/* Input: zmm0 = vector of a's zmm1 = vector of b's Output: zmm3 = vector of a/b */	/* Input: zmm0 = vector of a's zmm1 = vector of b's Output: zmm2 = vector of a/b */

Example 17-42. Dividing Packed Doubles, 26 and 14 Bits (Contd.)

Double Precision, Divide, 26 Bits	Double Precision, Divide, 14 Bits

__asm {
  vrcp14pd zmm2, zmm1
  vmulpd zmm3, zmm0, zmm2
  vmovapd zmm4, zmm0
  vfnmadd231pd zmm4, zmm3, zmm1
  vfmadd231pd zmm3, zmm4, zmm2
}
	__asm {
	  vrcp14pd zmm2, zmm1
	  vmulpd zmm2, zmm0, zmm2
	}

Example 17-43. Reciprocal Square Root of Doubles, 51 Bits

Double Precision, Reciprocal Square Root, 51 Bits

/* Input:
  zmm0 = vector of a’s
  zmm1 = vector of 1’s
  Output:
  zmm0 = vector of 1/sqrt (a)
*/
__asm {
  vsqrtpd zmm0, zmm0
  vdivpd zmm0, zmm1, zmm0
}

Example 17-44. Reciprocal Square Root of Doubles, 52 and 50 Bits

Double Precision, Reciprocal Square Root, 52 Bits	Double Precision, Reciprocal Square Root, 50 Bits

/* Input:
zmm4 = vector of a's
Output:
zmm0 = vector of 1/sqrt (a)
*/
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};
__asm {
vbroadcastsd zmm4, big_num
vmovapd zmm0, one
vmovapd zmm5, dc1
vmovapd zmm6, dc2
vmovapd zmm7, dc3
 	
vrsqrt14pd zmm3, zmm4
vfpclasspd k1, zmm4, 0x5e
vmulpd zmm1, zmm3, zmm4, {rn-sae}
vfnmadd231pd zmm0, zmm3, zmm1
vfmsub231pd zmm1, zmm3, zmm4, {rn-sae}
vfnmadd213pd zmm1, zmm3, zmm0
vmovups zmm0, zmm7
vmulpd zmm2, zmm3, zmm1
vfmadd213pd zmm0, zmm1, zmm6
vfmadd213pd zmm0, zmm1, zmm5
vfmadd213pd zmm0, zmm2, zmm3
vorpd zmm0{k1}, zmm3, zmm3
}
	/* Input:
	zmm3 = vector of a's
	Output:
	zmm4 = vector of 1/sqrt (a)
	*/
	// duplicates x eight times
	#define DUP8_DECL(x) x, x, x, x, x, x, x, x
	// used for aligning data structures to n bytes
	#define ALIGNTO(n) __declspec(align(n))
	ALIGNTO(64) __int64 one[ ] =
	{DUP8_DECL(0x3FF0000000000000)};
	ALIGNTO(64) __int64 dc1[ ] =
	{DUP8_DECL(0x3FE0000000000000)};
	ALIGNTO(64) __int64 dc2[ ] =
	{DUP8_DECL(0x3FD8000004600001)};
	ALIGNTO(64) __int64 dc3[ ] =
	{DUP8_DECL(0x3FD4000005E80001)};
	__asm {
	vmovapd zmm5, one
	vmovapd zmm6, dc1
	vmovapd zmm8, dc3
	vmovapd zmm7, dc2
	
	vrsqrt14pd zmm2, zmm3
	vfpclasspd k1, zmm3, 0x5e
	vmulpd zmm0, zmm2, zmm3, {rn-sae}
	vfnmadd231pd zmm0, zmm2, zmm5
	vmulpd zmm1, zmm2, zmm0
	vmovapd zmm4, zmm8
	vfmadd213pd zmm4, zmm0, zmm7
	vfmadd213pd zmm4, zmm0, zmm6
	vfmadd213pd zmm4, zmm1, zmm2
	vorpd zmm4{k1}, zmm2, zmm2
	}

Example 17-45. Reciprocal Square Root of Doubles, 26 and 14 Bits

Double Precision, Reciprocal Square Root, 26 Bits	Double Precision, Reciprocal Square Root, 14 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm1 = vector of 1/sqrt (a)
*/
 	
double half = 0.5;
 	
__asm {
vrsqrt14pd zmm1, zmm0
vmulpd zmm0, zmm0, zmm1
vbroadcastsd zmm3, half
vmulpd zmm2, zmm1, zmm3
vfnmadd213pd zmm2, zmm0, zmm3
vfmadd213pd zmm1, zmm2, zmm1
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm2 = vector of 1/sqrt (a)
	*/
	
	__asm {
	vrsqrt14pd zmm2, zmm0
	}

Example 17-46. Square Root of Packed Doubles, 53 and 52 Bits

Double Precision, Square Root, 53 Bits (IEEE)	Double Precision, Square Root, 52 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm2 = vector of sqrt (a)
*/
 	
__asm {
vsqrtpd zmm2, zmm0
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm0 = vector of sqrt (a)
	*/
	
	double half = 0.5;
	
	__asm {
	vbroadcastsd zmm4, half
	vrsqrt14pd zmm1, zmm0
	vfpclasspd k2, zmm0, 0xe
	vmulpd zmm2, zmm0, zmm1, {rn-sae}
	vmulpd zmm1, zmm1, zmm4
	knotw k3, k2
	vmovapd zmm3, zmm4
	vfnmadd231pd zmm3, zmm1, zmm2, {rn-sae}
	vfmadd213pd zmm2, zmm3, zmm2, {rn-sae}
	vfmadd213pd zmm1, zmm3, zmm1, {rn-sae}
	vfnmadd231pd zmm0 {k3}, zmm2, zmm2, {rn-sae}
	vfmadd213pd zmm0 {k3}, zmm1, zmm2
	}

Example 17-47. Square Root of Packed Doubles, 26 and 14 Bits

Double Precision, Square Root, 26 Bits	Double Precision, Square Root, 14 Bits

/* Input:
zmm0 = vector of a's
Output:
zmm0 = vector of sqrt (a)
*/
 	
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
 	
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
 	
ALIGNTO(64) __int64 OneHalf[ ] =
{DUP8_DECL(0X3FE0000000000000)};
__asm {
vrsqrt14pd zmm1, zmm0
vfpclasspd k2, zmm0, 0xe
knotw k3, k2
vmulpd zmm0 {k3}, zmm0, zmm1
vmulpd zmm1, zmm1, ZMMWORD PTR [OneHalf]
vfnmadd213pd zmm1, zmm0, ZMMWORD PTR
 	
[OneHalf]
vfmadd213pd zmm0 {k3}, zmm1, zmm0
}
	/* Input:
	zmm0 = vector of a's
	Output:
	zmm0 = vector of sqrt (a)
	*/
	
	__asm {
	vrsqrt14pd zmm1, zmm0
	vfpclasspd k2, zmm0, 0xe
	knotw k3, k2
	vmulpd zmm0 {k3}, zmm0, zmm1
	}

17.26 CLDEMOTE

Using the CLDEMOTE instruction, a processor puts a cache line into the last shared level of the cache hierarchy so that other CPU cores 'find' the same cache line in the last shared level and expensive cross-core snoop is avoided. The most significant advantage of CLDEMOTE is that multiple consumers can access the shared cache line amortizing each snoop request portion.

17.26.1 PRODUCER-CONSUMER COMMUNICATION IN SOFTWARE

In a multiprocessor environment, data sharing between the producers and consumers is an undisputed event. A cache hierarchy solves the major problem of accessing the line from the main memory resulting in faster data transfers. Typical cache hierarchy contains:

- Private L1 data and L1 instruction cache.
- A shared L2 cache for sibling hardware thread.
- A common L3 cache for all the CPU cores.

When a producer consumes data from the I/O or produces it, it is brought into the producer's L1 cache. Consumers read the data by initiating read requests, translating it into cross-core snoops, request, and response events. Consumers report L3 cache miss events and producer cores responding to the consumer core's snoop request. Multiplexing these cross-cores requests and responses when dealing with multiple consumers is detrimental.

17.27 TIPS ON COMPILER USAGE

This section explains some of the important compiler options that can be used with the Intel compiler to derive the best performance on a Skylake server.

For complete information on the compiler options and tuning tips, see the Intel Resource & Documentation Center.

Many options have names that are the same on Linux* and Windows*, except that the Windows* form starts with an initial Q. Within text, such option names are shown as [Q]option-name.

The default optimization level is O2 (unless -g is specified, in which case the default is O0). Level O2 enables many compiler optimizations including vectorization. Optimization level O3 is recommended for loop-intensive and HPC applications, as it enables more aggressive loop and memory-access optimizations, such as loop fusion and loop blocking to allow more efficient use of the caches.

- For the best performance of Skylake server microarchitecture, applications should be compiled with the processor-specific option [Q]xCORE-AVX512.
  - Notably, an executable compiled with these options will not run on non-Intel processors or on Intel processors that support only lower instruction sets.
- For users who want to generate a common binary that can be executed on Skylake server microarchitecture and the Intel® Xeon Phi™ processors based on Knights Landing microarchitecture, use the option [Q]xCOMMON-AVX512.
  - This option has a performance cost on both Skylake server microarchitecture and Intel® Xeon Phi™ processors compared with executables generated with the target-specific options:
    - [Q]xCORE-AVX512 on Skylake server
    - [Q]xMIC-AVX512 on Intel® Xeon Phi™ processors.

Additionally, users can tune the zmm code generation done by the compiler for Skylake server microarchitecture using the additional option **-qopt-zmm-usage=low|high** (/Qopt-zmm-usage:low|high on Windows).

- The argument value of low provides a smooth transition experience from Intel AVX2 ISA to Intel AVX-512 ISA on a Skylake server microarchitecture target, such as for enterprise applications.
  - Tuning for ZMM instruction use via explicit vector syntax such as #pragma omp simd simdlen() is recommended.
- The argument value of high is recommended for applications, such as HPC codes, that are bounded by vector computation to achieve more compute per instruction through use of the wider vector operations:
  - The default value is low for Skylake server microarchitecture-family compilation targets, such as [Q]xCORE-AVX512.
  - High for CORE/MIC AVX512 combined compilation targets such as [Q]xCOMMON-AVX512.

It is also possible to generate a fat binary that supports multiple instruction sets by using the [Q]axtarget option.

For example, if the application is compiled with [Q]axCORE-AVX512,CORE-AVX2:

- The compiler might generate specialized code for the Skylake server microarchitecture and AVX2 targets, while also generating a default code path that will run on any Intel or compatible, non-Intel processor that supports at least Intel® Streaming SIMD Extensions 2 (Intel® SSE2).
  - At runtime, the application automatically detects whether it is running on an Intel processor.
    - If so, it selects the most appropriate code path for Intel processors;
    - if not, the default code path is selected.
  - Irrespective of the options used, the compiler might insert calls into specialized library routines, such as optimized versions of memset/memcpy, that will dispatch to the appropriate codepath at runtime based on processor detection.

The option **-qopt-report[n]** (/Qopt-report[:n] on Windows) generates a report on the optimizations performed by the compiler, by default it is written to a file with a .optrpt file extension. n specifies the level of detail, from 0 (no report) to 5 (maximum detail).

The option **-qopt-report-phase** (/Qopt-report-phase on Windows) controls report generation from various compiler phases, but it is recommended to use the default setting where the report is generated for all compiler phases.

- The report is a useful tool to gain insight into the performance optimizations performed, or not performed, by the compiler, and also to understand the interactions between multiple optimizations such as inlining, OpenMP* parallelization, loop optimizations (such as loop distribution or loop unrolling) and vectorization.
- The report is based on static compiler analysis. Hence the reports are most useful when correlated with dynamic performance analysis tools, such as Intel® VTune™ Amplifier or Vectorization Advisor (part of Intel® Advisor XE), that do hotspot analysis and provide other dynamic information.
- Once this information is available, the optimization information can be studied for hotspots (functions/loopnests) in compiler reports.
  - The compiler can generate multiple versions of loop-nests, so it is useful to correlate the analysis with the version actually executed at runtime.
    - The phase ordering of the compiler loop optimizations is intended to enable optimal vectorization. Often, understanding the loop optimization parameters helps to further tune performance.
    - Finer control of these loop optimizations is often available via pragmas, directives, and options.

If the application contains OpenMP pragmas or directives, it can be compiled with -qopenmp (/Qopenmp on Windows) to enable full OpenMP based multi-threading and vectorization. Alternatively, the SIMD vectorization features of OpenMP alone can be enabled by using the option -qopenmp-simd (/Qopenmp-simd on Windows).

For doing studies where compiler-based vectorization has to be turned off completely, use the options

-no-vec -no-simd -qno-openmp-simd (/Qvec- /Qsimd- /Qopenmp-simd- on Windows).

Data alignment plays an important role in improving the efficiency of vectorization. This usually involves two distinct steps from the user or application:

- Align the data.
  - When compiling a Fortran program, it is possible to use the option -align array64byte (/align:array64byte on Windows) to align the start of most arrays at a memory address that is divisible by 64.
  - For C/C++ programs, data allocation can be done using routines such as _mm_malloc(…, 64) to align the return-value pointer at 64 bytes. For more information on data alignment, see https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.
- Convey the alignment information to the compiler using appropriate clauses, pragmas, and directives.

Compiler-based software data prefetching can be enabled with the options -O3 -xcore-avx512 -qopt-prefetch[=n] (-O3 /QxCORE-AVX512 /Qopt-prefetch[=n] on Windows), for n=0 (no prefetching) to 5 (maximal prefetching). Using a value of n=5 enables aggressive compiler prefetching, disregarding any hardware prefetching, for strided loads/stores and indexed loads/stores which appear inside loops. Using a value of n=2 reduces the amount of compiler prefetching and restricts it only to direct memory accesses where the compiler heuristics determine that the hardware prefetcher may not be able to handle well. It is recommended to try values of n=2 to 5 to determine the best prefetching strategy for a particular application. It is also possible to use the -qopt-prefetch-distance=n1[,n2] (/Qopt-prefetch-distance=n1[,n2] on Windows) option to fine-tune application performance.

- Useful values to try for n1: 0,4,8,16,32,64.
- Useful values to try for n2: 0,1,2,4,8.

Loop-nests that have a relatively low trip-count value at runtime in hotspots can sometimes lead to sub-optimal AVX-512 performance unless the trip-count is conveyed to the compiler. In many such cases, the compiler will be able to generate better code and deliver better performance if values of loop trip-counts, loop-strides, and array extents (such as for Fortran multi-dimensional arrays) are all known to the compiler. If that is not possible, it may be useful to add appropriate loop_count pragmas to such loops.

Interprocedural optimization (IPO) is enabled using the option -ipo (/Qipo on Windows). This option can be enabled on all the source-files of the application or it can be applied selectively to the source files containing the application hot-spots. IPO permits inlining and other inter-procedural optimizations to happen across these multiple source files. In some cases, this option can significantly increase compile time and code size. Using the option -inline-factor=n

(/Qinline-factor:n on Windows) controls the amount of inlining done by the compiler. The default value of n is 100, indicating 100%, or a scale factor of 1. For example, if a value of 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2, thus enabling more inlining than the default.

Profile-guided optimizations (PGO) are enabled using the options -prof-gen and -prof-use (/Qprof-gen and /Qprof-use on Windows). Typically, using PGO increases the effectiveness of using IPO.

The option -fp-model name (/fp:name on Windows) controls tradeoffs between performance, accuracy and reproducibility of floating-point results at a high level. The default value for name is fast=1. Changing it to fast=2 enables more aggressive optimizations at a slight cost in accuracy or reproducibility. Using the value precise for name disallows optimizations that might produce slight variations in floating-point results. When name is double, extended or source, intermediate results are computed in the corresponding precision. In most situations where enhanced floating-point consistency and reproducibility are needed -fp-model precise -fp-model source (/fp:precise /fp:source on Windows) are recommended.

The option -fimf-precision=name (/Qimf-precision=name on Windows) is used to set the accuracy for math library functions. The default is OFF, which means that the compiler uses its own default heuristics. Possible values of name are high, medium, and low. Reduced precision might lead to increased performance and vice versa, particularly for vectorized code. The options -[no-]prec-div and -[no-]prec-sqrt improve[reduce] precision of floating-point divides and square root computations. This may slightly degrade [improve] performance. For more details on floating-point options, see Consistency of Floating-Point Results using the Intel® Compiler (2018) .

The option -[no-]ansi-alias (/Qansi-alias[-] on Windows) enables [disables] ANSI and ISO C Standard aliasing rules. By default, this option is enabled on Linux, but disabled on Windows. On Windows, especially for C++ programs, adding /Qansi-alias to the compilation options enable the compiler to perform additional optimizations, particularly taking advantage of the type-based disambiguation rules of the ANSI Standard, which says for example, that pointer and float variables do not overlap.

If the optimization report specifies that compiler optimizations may have been disabled to reduce compile-time, use the option -qoverride-limits to override such disabling in the compiler and ensure optimization is applied. This can sometimes be important for applications, especially ones with functions that have big bodies. Note that using this additional option may increase compile time and compiler memory usage significantly in some cases.

The list below shows a sampling of loop-level controls available for fine-tuning optimizations - including a way to turn off a particular transformation reported by the compiler.

- #pragma simd reduction(+:sum)
  - The loop is transformed as is, no other loop-optimizations will change the simd-loop.
- #pragma loop_count min(220) avg (300) max (380)
  - Fortran syntax: !dir$ loop count(16)
- #pragma vector aligned nontemporal
- #pragma novector // to suppress vectorization
- #pragma unroll(4)
- #pragma unroll(0) // to suppress loop unrolling
- #pragma unroll_and_jam(2) // before an outer loop
- #pragma nofusion
- #pragma distribute_point
  - If placed as the first statement right after the for-loop, distribution will be suppressed for that loop.
  - Fortran syntax: !dir$ distribute point
- #pragma prefetch *:<hint>:<distance>
  - Apply uniform prefetch distance for all arrays in a loop.
- #pragma prefetch <var>:<hint>:<distance>

  - Fine-grained control for each array
- #pragma noprefetch [<var>]
  - Turns off prefetching [for a particular array]
- #pragma forceinline (recursive)
  If placed before a call, this is a hint to the compiler to recursively inline the entire call-chain.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimization in the product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

CHAPTER 18
INTEL® ADVANCED VECTOR EXTENSIONS 512 - FP16 INSTRUCTION SET FOR INTEL® XEON® PROCESSORS

18.1 INTRODUCTION

The Intel® AVX-512 FP16 instruction set architecture supports a wide range of general purpose numeric operations for 16-bit half-precision IEEE-754 floating-point and complements the existing 32-bit and 64-bit floating-point instructions already available in Intel Xeon processors. This instruction set architecture also provides complex-valued native hardware support.

This instruction set architecture is ideal for numeric operations where reduced precision can be used, such as signal and media processing. For example, wireless signal processing operations such as beam-forming, precoding, and minimum mean squared error (MMSE) perform well with this ISA. Furthermore, traditional signal processing such as real or complex-valued fast Fourier transform (FFTs) also works well with this instruction set. The advantage of using reduced precision in these cases is that because fewer bits are processed for each element, the overall compute throughput can be increased, allowing precision and speed to be traded against each other.

18.2 TERM DEFINITIONS

Table 1-2 provides definitions of terminology used throughout this volume.

18.3 OVERVIEW

In this chapter, we describe the addition of the FP16 ISA for Intel AVX-512 into the Intel Xeon processor family to handle IEEE 754-2019-compliant half-precision floating-point operations (also known officially as binary16, or unofficially as FP16). This instruction set is general-purpose and can be used for all numeric operations that could be reasonably expected, including numeric operations (add, subtract, divide, multiply), fused operations (for example, fused multiply-add), comparisons, conversions to and from other data types, and many more. Broadly, the FP16 instruction set mirrors the floating-point support that is already available in Intel Xeon processors for 32-bit (FP32) and 64-bit (FP64), although there are a few exceptions to this, which will be noted where appropriate. There is one notable new feature of FP16 when compared to existing FP32 and FP64 instruction sets: the addition of native complex-value support for interleaved FP16 data, which is useful in scientific computing and signal processing.

The two major advantages of using the FP16 instruction set compared to other floating-point formats are increased execution throughput and reduced storage requirements. Half-precision floating-point values only require 16 bits for storing each value, as opposed to the 32 or 64 bits needed for other common IEEE floating-point formats. This allows FP16 to handle twice as many operations per each clock cycle compared to FP32, and four times as many compared to FP64. Similarly, the reduced size means that more values can be stored in a given memory region compared to the other formats, increasing the effectiveness of the registers and the cache hierarchy. The disadvantages are the reduced range and precision. It is the responsibility of the programmer to decide whether this floating-point format is suitable for a certain application.

Half-precision floating-point is useful for building systems where the dynamic range of floating-point is required but a lower numeric precision can be easily tolerated and traded for higher compute performance. Typical applications for half-precision floating-point include signal processing, media or video processing, artificial intelligence, and machine learning.

Historically, some limited support for half-precision data types was available in processors from the 3rd generation Intel® Core™ processor onwards, but the operations were restricted to converting between half-precision and FP32 values. On older platforms, all numeric operations had to be implemented using higher precision formats and down-converted on completion. Those instructions were useful for compatibility with other platforms (for example, Intel® GPUs), but did not realize the benefits in higher compute performance brought about in FP16.

IEEE FP16 is not the only 16-bit floating-point format. Another common type is bfloat16, which is primarily used in artificial intelligence and machine learning. Intel Xeon processors support some bfloat16 operations, including type conversions and a few limited numeric operations, but not the full range of general-purpose operations that are supported in FP16 for Intel AVX-512. This chapter describes only the instruction set relating to IEEE 754-2019.

This chapter covers both the general-purpose instruction set as well as the new complex-valued instructions. We then look at the numeric implications of using FP16 and discuss how to write optimal code sequences for some common operations.

The examples provided in this document use the intrinsic and data type support provided as part of the Intel® OneAPI DPC++ Compiler.

18.4 FP16 NUMERIC INSTRUCTIONS

FP16 is an instruction set extension that mirrors the existing support for other floating-point operations in Intel AVX-512 and makes it available in IEEE-754 FP16 (binary16) number format. It is a general-purpose instruction set, and features instructions that support all common operations that are required in typical numeric software applications. Briefly, the following classes of instructions are supported:

- Fundamental IEEE numeric: Addition, subtraction, multiplication, division, and square root.
- Fused: Fused (multiply-accumulate) operations covering fmadd, fmsub, negated fma, fmaddsub, and fmsubadd.
- Comparison: Minimum, maximum, and compare-to-mask (e.g., neq, lt, gt, etc.).
- Conversions: Conversions to and from other common data types, including 16/32/64-bit integer and FP32/FP64 floating-point.
- Approximation: Fast, but approximate operations to support reciprocal and reciprocal-square-root.
- Specialized: Significand (mantissa)/exponent manipulation, scaling, and rounded scaling.
- Complex: Native complex-value multiply and fused-multiply operations.

The following sections will provide information on how to use these new instructions, the impact on performance, and the consequences of the reduced floating-point decision.

18.4.1 DATA TYPE SUPPORT

Table 18-1 shows the new data types supported with the FP16 instruction set. In each case, the name of the equivalent type in C or C++ is provided.

Table 18-1. Supported FP16 Data Types

Type Format	C/C++ Type Name	Notes
Scalar	_Float16	Single 16-bit value stored in IEEE FP16 format
128-bit AVX register	__m128h	8 x FP16 values, or 4 x complex FP16 values (CFP16)
256-bit AVX2 register	__m256h	16 x FP16 values, or 8 x complex FP16 values (CFP16)
512-bit AVX-512 register	__m512h	32 x FP16 values, or 16 x complex FP16 values (CFP16)

The complex instructions operate on standard SIMD vector types, such as __m128h, but internally those instructions treat the register as sets of complex-valued pairs, as shown in Figure 18-1. We refer to a complex pair of FP16 values as `CFP16'. The CFP16 type is laid out as though it were an array of two FP16 values, or a C++ type such as std::complex<_Float16>.

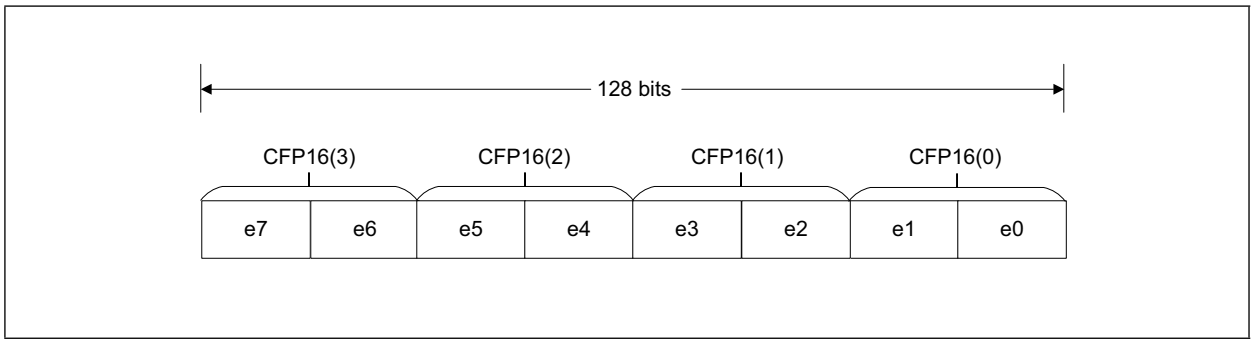


Figure 18-1. Layout of a 128-Bit Register Representing Four Complex FP16 (CFP16) Values

In the latest Intel OneAPI compilers, 16-bit floating-point literals can be created by suffixing a value with f16. For example:

_Float16 value = 12.34f16;

18.4.2 OVERVIEW OF INTRINSICS

In common with the intrinsics for other vector instruction sets, FP16 intrinsics take the following form:

result = _mmBITLENGTH_OPNAME_ELEMENTTYPE(arguments)

The bit-length can be 512, 256, or 128 bits, and in the 128-bits case the BITLENGTH field is empty. The OPNAME is a short descriptor or abbreviation of what the operation does (for example, add, sub, fmadd). The element type is sh for FP16 scalar (scalar-half), ph for a vector of FP16 values (packed-half), or pch for a vector of CFP16 values (packed-complex-half). Table 18-2 gives some examples to illustrate the naming conventions.

Table 18-2. Example Intrinsic Names

Intrinsic Name	Description
_mm_sub_sh	Subtract a single scalar FP16 element from another scalar FP16 element.
_mm_add_ph	Add a pair of 8xFP16 vector registers to form a result containing 8xFP16 outputs.
_mm256_add_ph	Add a pair of 16xFP16 vector registers to form a result containing 16xFP16 outputs.
_mm512_add_ph	Add a pair of 32xFP16 vector registers to form a result containing 32xFP16 outputs.
_mm256_fmadd_ph	Multiply a pair of 16xFP16 vector registers and add the result to a third vector register of 16xFP16 values, forming a result containing 16xFP16 vector elements.
_mm512_rcp_ph	Compute the reciprocal of a vector register containing 32xFP16 values, generating an output of another vector register containing 32xFP16 values.
_mm256_fmadd_pch	Compute the complex multiplication of 8xCFP16 (complex-FP16) values, adding the result to another such register, and generating a result containing 8xCFP16 elements.

Table 18-2. Example Intrinsic Names

Intrinsic Name	Description
_mm512_conj_pch	Compute the conjugate of a 512-bit register containing 16xCFP16 (complex-FP16) elements.

Note that pch complex operation intrinsics are only provided for multiply and fused-multiply-add operations since these require special hardware support. No intrinsics are provided for operations like addition, since the existing **add_ph** intrinsic behaves correctly for those without extra support requirements.

For a complete list of all the intrinsics provided as part of FP16, refer to the Intel® Intrinsics Guide.

In the remainder of this chapter where the names of intrinsics are given, typically only the 128-bit variant is shown. Because AVX-512-FP16 supports VL encoding, all three length variants of the intrinsics are available (i.e., 128-bit, 256-bit, 512-bit).

18.4.3 FUNDAMENTAL COMPLEX-VALUED SUPPORT

For complex-valued operations the primary place where hardware support is provided is in multiplication. Complex multiplication requires several steps, and the FP16 ISA accelerates those steps. Simpler operations, such as addition and subtraction, do not require explicit complex support since these can be handled using the other FP16 instructions (for example, addition of two complex numbers is just the addition of respective real and imaginary values from the two inputs, so _mm_add_ph can be directly used). Note that complex division is not supported in hardware as this is an uncommon operation, and it can be constructed from the hardware multiplier and complex multiplier support if required.

To illustrate the mechanics of how complex multiplication is supported, consider the following complex multiply:

(a + bi) * (c + di)

This operation can be refactored as follows:

(ac - bd) + (ad + bc)i

Note that to compute each of the real and imaginary components of the multiply a stand-alone multiply is used first, followed by a suitable fmadd/fmsub instruction. The hardware support for complex-valued multiplies uses these partial mul/fmadd instructions in sequence to perform the entire complex multiply. The hardware can schedule and route the data inside the processor to do this more quickly and efficiently than using an explicit instruction sequence to move the real and imaginary data into the correct places. Note however that each intermediate step produces a temporary FP16 answer, so the final result will have had an FP16 quantization step.

Using the symbols from the example above, a complex-fma (that is, accumulating against another complex number) can be implemented using the following refactoring:

((accReal + ac) - bd) + ((accImag + ad) + bc)i

This sequence is equivalent to two FMA operations being performed for each of the real/imag components.

The conjugate of a complex number is formed by negating its imaginary component. A common operation with conjugation is to multiply a complex number with a conjugate of another complex number. Conjugation in FP16 is supported using three classes of intrinsic, as illustrated in Table 18-3.

Table 18-3. Conjugation Instructions

Intrinsic Name	Description
_mm_conj_pch	Compute the conjugate of a register containing CFP16 (complex-FP16) elements by negating each imaginary value.
_mm_fcmul_pch	Compute the multiplication of a conjugated value with another complex value.
_mm_fcmadd_pch	Compute the multiplication of a conjugated value with another complex value, adding the result to a third complex-vector register.

Both the multiply and the FMA are able to perform the conjugation as part of the instruction operation itself. It is not necessary to conjugate the value first. For example, an **_mm_fcmul_pch** intrinsic is functionally equivalent to:

_mm_fcmul_pch(_mm_conj_pch(lhs), rhs)

However, **_mm_fcmul_pc** will operate in fewer cycles than calling that sequence explicitly. When the compiler notices separate conjugate and multiply intrinsics being used, it fuses them into a single conjugate-multiply.

18.4.4 USING INTEL® AVX-512 BIT MASKS FOR REAL-VALUED OPERATIONS

FP16 is able to use the bit-mask features of Intel AVX-512 both to control when operations in a vector register take place, and to generate masks that store the results of performing tests on vector registers.

Masks allow execution of an instruction to be conditionally applied to selected elements of a vector register. Most instructions permit such a mask register to be supplied as part of the operation, where each bit within the mask corresponds to a different element of the vector register. If a given mask bit is set, then the instruction operates on the corresponding element. If the bit is cleared, the operation is not performed and that element's output is replaced by another value. The cleared output value can either be taken from another source register, or it can be zeroed. The operation of a masked instruction is illustrated in Figure 18-2. Note that the operation only takes place where the 8-element mask has a corresponding bit set and all other outputs are zeroed.

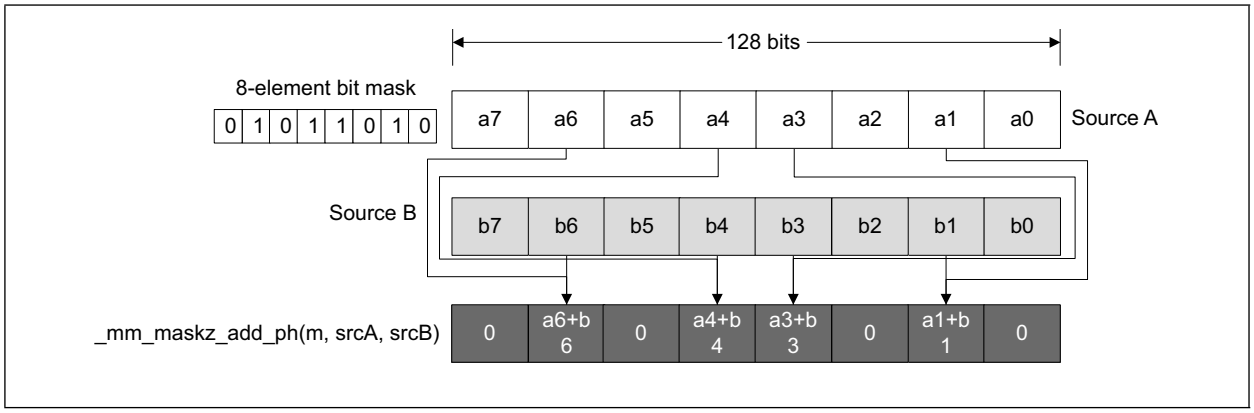


Figure 18-2. A Zero-Masked FP16 Add On Two 128-Bit Vectors

Note that masks also control whether faults within an instruction are suppressed. If an operation generates a fault in a particular element, but the element's operation has been disabled by a zero bit in the mask, then the fault is not reported.

Some instructions in Intel AVX-512 can generate mask registers, and with FP16, these are normally the result of a comparison operation. For example, consider the following code snippet:

```
whichElementsAreLess = _mm512_cmp_ph_mask(lhs, rhs, _CMP_LT_OS);
```

In this example, every element of the left-hand vector is compared to see if it is less than the corresponding element in the right-hand vector. If the left element is less than the right element, then a 1 is generated in the mask bit output, otherwise a zero is emitted. This comparison instruction allows all the major binary comparison operations to be performed between two vectors.

The FP16 instruction set also provides support to test for special values using the **_mm_fpclass_ph_mask** instruction. This instruction takes a special immediate value that directs the instruction to the numeric classes to look for in the vector register (for example, infinities, NaN, zero, denormal). This instruction is often used in combination with other instructions to remove special case values from a register and replace them with something different. For example, the following code snippet removes NaN values and replaces them with zero:

```
__mmask8 whichAreNan = _mm_fpclass_ph_mask(values, QUIET_NAN | SIGNAL_NAN);
__m128h valuesWithNoNan = _mm_mask_blend_ph(whichAreNan, values, __m128h());
```

In Intel AVX-512, there is a special instruction that does direct replacement of special values with known constants called **_mm_fixupimm_ps/pd**, but this is unavailable in the FP16 instruction set.

18.4.5 USING INTEL® AVX-512 BIT MASKS FOR COMPLEX-VALUED OPERATIONS

When a mask operation is applied to an intrinsic that operates directly on complex instruction data (for example, **_mm512_mask_fmadd_pch)**, then each mask bit refers to a complex pair of FP16 values, not to the individual FP16 values. This is illustrated in Figure 18-3. Note that there are eight FP16 elements, grouped into four CFP16 complex values. The four mask bits correspond to the four CFP16 values.

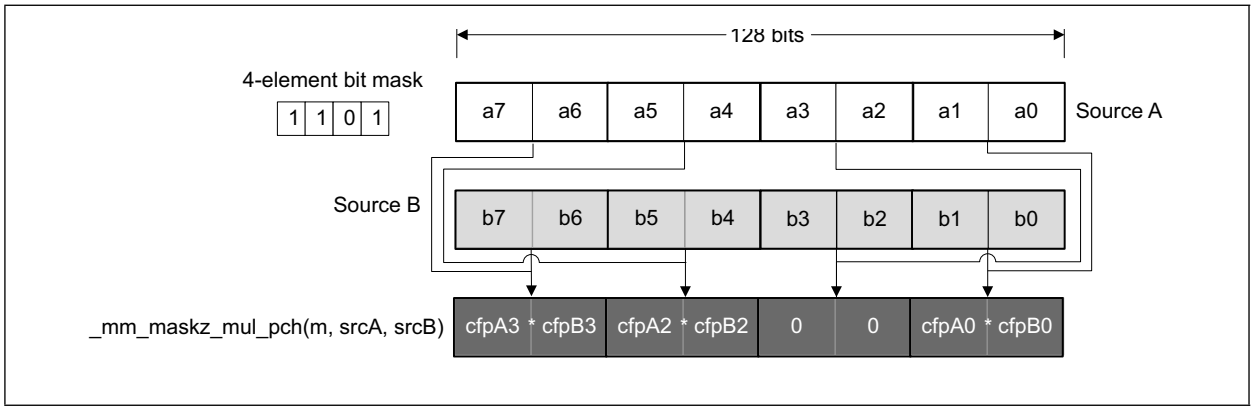


Figure 18-3. A Masked Complex Multiplication

No direct numeric support is provided for complex operations such as addition, subtraction, and real-valued scaling, but their standard real-valued equivalent instructions can be used instead. However, if such an operation has to be masked on a per-complex-element basis, then the incoming complex-valued mask needs to be expanded into pairs of identical bits, one pair per complex-element. An example of this is illustrated in Figure 18-4. Note that the incoming mask bit, which is per CFP16 element, needs to be expanded to duplicate each bit for the real-valued intrinsic.

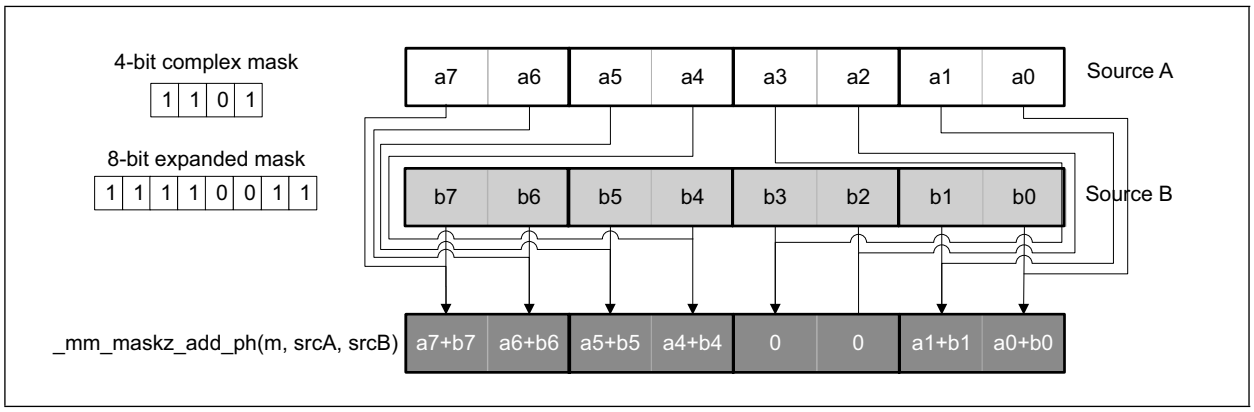


Figure 18-4. Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition

The operation to expand the incoming complex-mask-bits to generate real-valued mask could be performed in numerous different ways, but one efficient way to achieve this operation is shown in Example 18-1. This code fragment uses the fast mask-to-vector and vector-to-mask instructions to effect the upscaling of the bit-mask elements.

Example 18-1. Converting a Complex-Valued Mask to a Real-Valued Mask by Duplicating Adjacent Bits

```
__mmask8 getRealMaskFromComplexMask(__mmask8 m)
{
    // 4 incoming bits representing the 4 complex elements in a 128-bit register.
    // Each mask bit is converted into an entire element in a vector register
    // where a 0-mask generates 32x0, and a 1-mask generates 32x1. For example
    // 0010 -> [000....00], [000...000],  [111....111], [000....000]
    auto wholeElements = _mm_movm_epi32(m);

    // Each complex element can now be treated as a pair of 16-bit elements instead,
    // and the MSB of each 16-bit unit can extracted as a mask bit in its own right.
    return _mm_movepi16_mask(wholeElements);
}
```

It may also be necessary to perform a similar operation in reverse, where pairs of bits representing adjacent FP16 values need to be reduced in some way into a single bit representing the complete complex element (e.g., AND, OR). For example, if two complex vectors must be compared for equality then the individual FP16 elements must be compared for equality first, and then if two adjacent mask bits are both set (that is, the logical AND of those bits), then the complex element as a whole must be equal. This comparison test is illustrated in Figure 18-5. Note that some of

the sub-elements in each CFP16 do compare equal when using the **_mm_cmp_ph_mask** intrinsic, but both elements in each CFP16 value must be equal for the complex values to be truly equal.

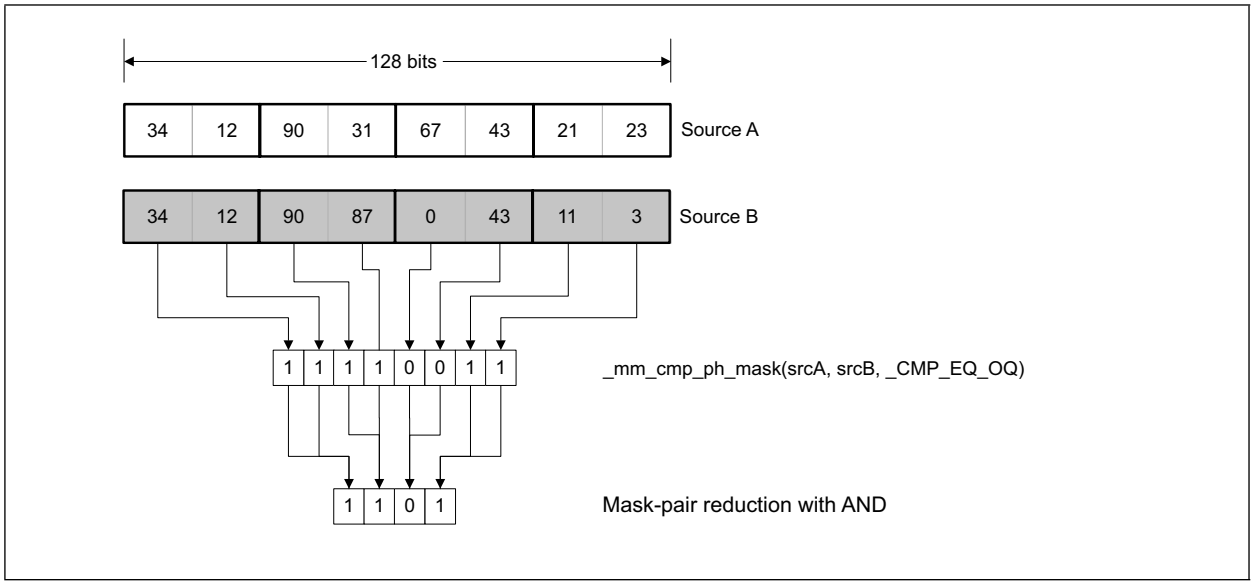


Figure 18-5. Comparison Operation Between Two Complex-Valued Vectors

One implementation of the function to combine adjacent mask bits using an AND operation is shown in Example 18-2. Like the example above, it uses the mask-to-vector and vector-to-mask instructions to good effect.

Example 18-2. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask By AND-Combining Adjacent Bits

```
__mmask8 getComplexMaskFromRealMask_AND(__mmask8 m)
{
    // 8 incoming bits representing the 8 real-valued elements in a 128-bit register.
    // Broadcast the bits into 8-bit elements of all 1's or all 0's.
    auto wholeElements = _mm_movm_epi8(m);

    // Generate an entire vector of 1's (typically a ternlogic will be used, which is
    // very cheap and can be done in parallel with the movm above, or hoisted when
    // used repeatedly.
    const auto allOnes = _mm_set1_epi16(-1);

    // Extract single mask bits from each 16-bit element which are the logical ANDs of the
    // MSBs of each incoming 8-bit element. Because the movm above generated all 0/1 bits
    // across the whole element the only combinations of values in each 32-bit unit are
    // both all zero, both all one, or one of each. The logical AND of the MSBs can only
    // occur when both 8-bit sub-elements are all ones, so this is equivalent to
    // comparing the 16-bit block as though it were entirely 1, which is a direct
    // equality comparison.
    return  _mm_cmp_epi16_mask(wholeElements, allOnes, _MM_CMPINT_EQ);
}
```

Note that the individual mask bits are expanded to 8-bit elements and then compared for equality as 16-bit elements to combine adjacent elements. There is no need to expand to the same size as the data being processed (that is, 16/32-bit respectively in this case), since the bitwise pairing is independent of the original data element sizes. By using smaller registers, efficiency is very slightly improved compared to using wider registers.

The adjacent mask bits could also be combined using an OR operation, which might be useful if testing whether a complex value is NaN (that is, a complex value is NaN if either of its individual elements is NaN). A sequence for determining an OR of adjacent mask bits is shown in Example 18-3.

Example 18-3. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask by OR-Combining Adjacent Bits

```
__mmask8 getComplexMaskFromRealMask_OR(__mmask8 m)
{
    auto wholeElements = _mm_movm_epi16(m);

    // Similar logic to the AND variant above but now any 32-bit element which
    // isn't zero represents the logical OR or two adjacent 16-bit block
    // elements in one 32-bit block.
    return  _mm_cmp_epi32_mask(wholeElements, __m128i(), _MM_CMPINT_NE);
}
```

18.5 NUMERICS

Using FP16 instead of the more conventional and widely used FP32 and FP64 formats introduces a number of interesting numeric behaviors. It is beyond the scope of this chapter to discuss these fully or to describe the numeric methods required to build FP16 algorithms, but in this section, we highlight a few of the properties and behaviors of the FP16 number format and the consequences that arise from this.

18.5.1 INTRODUCTION TO FP16 NUMBER FORMAT

An FP16 floating-point number is represented using 16 bits, which are laid out as shown in Figure 18-6. The figure also shows two other floating-point number formats for comparison, one being the common 32-bit FP32 format, and the other being the alternative 16-bit floating-point format called brain-float 16, which is used for machine learning. Note how bfloat16 is simply the upper 16-bits of the FP32 format, giving it the same dynamic range as FP32 but with considerably reduced precision, making this ideal for machine-learning applications. In contrast, the IEEE FP16 format modifies the sizes of the significand and the exponent to produce a more balanced blend of precision and range, which is more suitable for general purpose algorithms.

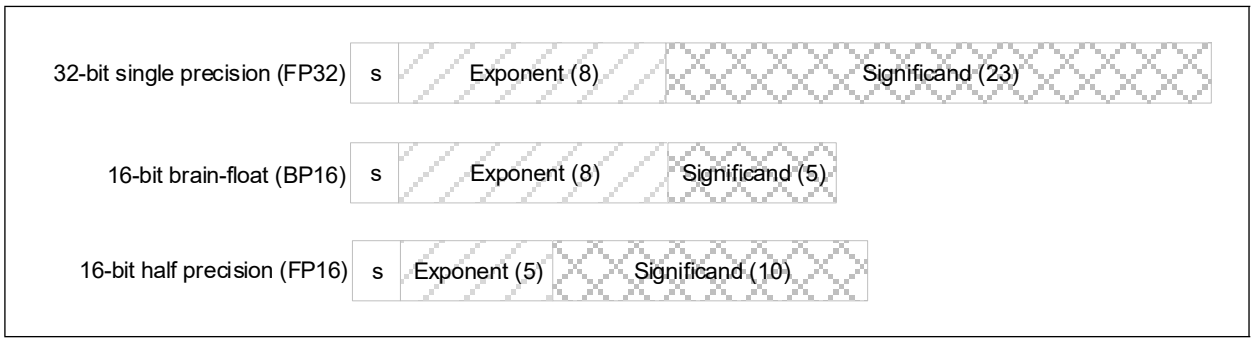


Figure 18-6. Bit Layout of Three Types of Floating-Point Formats

Certain bitwise operations can be used to manipulate the floating-point numbers without requiring special hardware support. For example, an absolute operation (that is, convert the value to its positive equivalent) can be implemented as a bitwise AND of the lower 15 bits of the value, thereby stripping off any sign bits. Similarly, functions like negate, negative-absolute (nabs), copy-sign, test-sign, and so on can also be implemented using existing Intel AVX-512 bitwise intrinsics.

18.5.2 OBSERVATIONS ON REPRESENTING NUMBERS IN FP16 FORMAT

Although FP16 behaves functionally the same way as FP32 and FP64, the limited number of bits in its representations means that some limits are imposed on the permitted values. In FP32 and FP64, most of the useful human-comprehensible numbers can be easily represented without considering too much about the limitations in value representation imposed by the floating-point format, but those limitations show up in FP16 limitations more easily. In Figure 18-7 some landmark values on the real-valued positive number line are illustrated, and in Table 18-4 further useful numbers are listed.

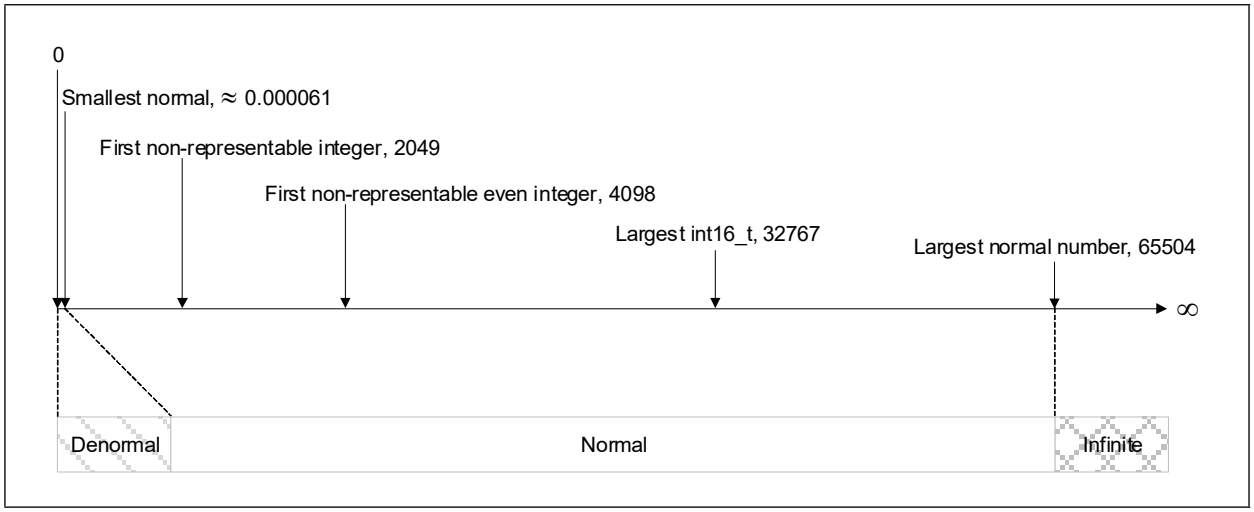


Figure 18-7. Landmark Numbers on the Real-Valued FP16 Axis

Table 18-4. Useful or Interesting FP16 Numbers

Value	Hex Representation	Description
0	0x0000	Zero
0.000000059604645	0x0001	Smallest denormal value
0.000060975552	0x03ff	Largest denormal value
0.00006103515625	0x0400	Smallest normal value
1	0x3c00	One
Inf	0x7c00	Positive Infinity
-Inf	0xfc00	Negative Infinity

Some consequences of using the FP16 number format include:

- Denormal numbers are not very small; it is easy to generate a number that gets too small to be represented using an FP16 normal value.
- Infinity starts very low down; it is only slightly less than the maximum value of an unsigned 16-bit integer. Overflow of values converted from 16-bit integers can quickly lead to infinities.
- Only integers up to a magnitude of 2048 can be fully represented. Beyond that, the permitted integers become very sparse very quickly. Rounding from a real integer type to an FP16 value introduces large absolute integer errors if the integer is above 2048.

These limitations may seem cumbersome at first, but there are good reasons why FP16 representation is a good fit for many signal-processing applications. Firstly, it is important to consider the Signal-to-Interference-plus-Noise ratio, or SINR. In a typical signal-processing system, such as a wireless receiver, the signals of interest are almost always subject to measurement noise. In the case of a wireless system, this noise would be introduced by receiver thermal-noise or in-band interference.

With FP16 number representation, any value on the real number-line within the normal range is subject to approximately -73.7 dB of quantization noise when quantized to FP16 format[1]. To use common signal-processing parlance, the SINR is always ~73.7 dB, meaning that the quantizing error variance is $10^{-7.37}$ times the squared-magnitude of the signal. When compared to a typical received SINR of 25 dB, this means that the variance of the additional error introduced by the FP16 quantization is $10^{48.7/10}$, ~74,131 times lower than the measurement noise of the signal. In effect, it adds a negligible extra noise power.

Other signal processing requirements include dynamic range. The FP16 representation is able to maintain the 73.7 dB SINR over the complete dynamic range of a perfect 16-bit ADC. Care must be taken to exploit the floating-point aspect of FP16 and not directly convert integers to FP16, as squaring operations will likely result in “Inf”. However, this is easily overcome by converting 16-bit integers to FP16 and then scaling by a fixed constant: 1/256 is a good choice.

18.5.3 NUMERIC ACCURACY GUARANTEES

In any floating-point calculation it is impossible to give the result of every possible computation because not every value has a valid representation. The output has to be quantized to a nearby value which can be represented in that

1. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS-II: EXPRESS BRIEFS, VOL. 63, NO. 6, JUNE 2016: Quantization Noise Power Estimation for Floating-Point DSP Circuits Gabriel Caffarena, Senior Member, IEEE, and Daniel Menard, Member, IEEE: https://ieeexplore.ieee.org/document/7407669.

number format. If the result is not representable, the distance between the next lowest representable value and the next highest representable value is called the unit-in-last-place, or ULP (and less commonly but equivalently, the unit-of-least-precision), and the actual answer will lie somewhere between the two. When the output value is rounded up or down to the nearest representable value, it therefore follows that the error in that calculation is no more than 0.5 ULP.

In IEEE 754 floating-point arithmetic, the standard mandates that the result of any hardware implementation will generate a correctly rounded result that has no more than 0.5 ULP of error when rounding `to nearest' for the following operations:

- Addition
- Subtraction
- Multiplication
- Division
- Square root
- Fused multiply-add

When rounding up, down, or toward zero, the error is less than 1 ULP.

The fused multiply-add guarantees that the intermediate result of the multiplication is kept in a higher precision form internally before being added. This means that the result of an FMA operation can have less overall error than doing a sequence of individual multiply and add instructions.

The Intel AVX-512 FP16 instruction set is compliant with IEEE Standard 754-2019, and arithmetic operations on it are implemented in the spirit of the Standard (which does not require arithmetic operations for binary16). Consequently, all the operations listed above yield correctly rounded results. FP16 also contains a few instructions (not defined in IEEE 754-2019) that produce approximate results to within 0.5625 ULP error. These include:

- Reciprocal (rcp)
- Reciprocal square-root (rsqrt)

Further examination of these special cases is given in later sections.

Note also that complex multiplications (and fused multiplications) have an intermediate quantization to FP16 because, as described earlier, the hardware implements these operations as a sequence of FMAs. Each step of that sequence introduces quantizing, so the overall effect of the complete complex multiply has some small error.

18.5.4 HANDLING DENORMAL VALUES

FP16 differs from FP32 and FP64 floating-point by allowing computations involving denormals to be performed with no impact on cycle count. This is in contrast to FP32 and FP64 computation modes where handling denormals under some conditions can introduce cycle performance penalties. In FP32 and FP64 computations, when a denormal value is encountered the instruction might trap and call into a software routine to handle the computation instead, which increases the number of processor cycles required. Describing when this occurs and what the penalties would be is beyond the scope of this document. However, it is common to attempt to avoid denormal values in FP32 and FP64 computations where possible by modifying two FP execution flags:

- **DAZ.** Denormals Are Zero: Any denormal inputs are replaced by zero before use.
- **FTZ.** Flush To Zero: Any outputs that would be denormal are replaced by zero.

Since FP16 handles denormals at full speed, all FP16 computations ignore the DAZ and FTZ flags and modifying these flags has no impact on FP16 numeric behavior or performance.

18.5.5 EMBEDDED ROUNDING

In common with other Intel AVX-512 instructions, the FP16 instruction set allows the use of an instruction attribute called Static Rounding Mode. Rather than depending upon the contents of a global control register (called MXCSR) to set the floating-point rounding mode, most of the Intel AVX-512 FP16 instructions can override the rounding mode behavior for only that instruction. Some permitted rounding modes are shown in Table 18-5.

For convenience, intrinsics are provided to give easy access to embedded rounding modes. For example, the FP16 addition instruction can have the rounding mode controlled by using the following intrinsic:

```
__m128h _mm512_add_round_ph (__m128h a, __m128h b, int
rounding);
```

The third parameter, which supplies the rounding mode immediate, can be taken from the third column of Table 18-5, which describes four of the IEEE rounding modes and the required selector to invoke that behavior.

Table 18-5. Conjugation Instructions

IEEE 754-2019 Rounding Mode	Description	C Intrinsic Constant Selector
roundTiesToEven	Round toward nearest floating point, with ties to even	_MM_FROUND_TO_NEAREST_INT | _MM_FROUND_NO_EXC
roundTowardPositive	Round toward negative infinity	_MM_FROUND_TO_NEG_INF | _MM_FROUND_NO_EXC
roundTowardNegative	Round toward positive infinity	_MM_FROUND_TO_POS_INF | _MM_FROUND_NO_EXC
roundTowardZero	Round toward zero	_MM_FROUND_TO_ZERO | _MM_FROUND_NO_EXC

The following points should be noted:

- When a rounding mode is explicitly used then this implies that the `suppress-all-exceptions' flag is also set for that instruction. Therefore, an instruction that uses embedded rounding never raises a floating-point exception.
- The C intrinsic constant selector name **_MM_FROUND_TO_NEAREST_INT** is not ideal, but that name has been historically used for so long in all common compilers that it is difficult to change to something more meaningful.
- Embedded rounding is only permitted on full width AVX-512 intrinsics (e.g., **__mm512_OP_round_pX**) or scalar operations (e.g., **_mm_OP_round_sX**). It is not permitted on AVX-512 VL encoded 128-bit or 256-bit operations.

18.5.6 LEGACY FP16 DATA TYPE CONVERSION

Two older Intel instruction sets already supported FP16 values as a storage format and provided conversion instructions to and from other data types. For example:

- **_mm_cvtph_ps**: Convert from FP16 to FP32.
- **_mm_cvtps_ph:** Convert from FP32 to FP16.

These instructions were originally available in the FP16C ISA for 128-bit and 256-bit registers. The Intel AVX-512 FP16 ISA further extended these instructions to work with 512-bit registers, and also added the option to conditionally mask selected elements (for example, **_mm512_mask_cvtph_ps**).

These instructions do not have embedded broadcast modes. It is recommended that the newer conversion instructions described in the Section 18.5.7 be used instead.

18.5.7 FP16 CONVERSIONS TO AND FROM OTHER DATA TYPES

The Intel AVX-512 FP16 ISA contains a comprehensive set of instructions that convert to and from most of the other supported data types, with and without rounding.

Conversions from FP16 to other data types take the following intrinsic forms:

- **_mm_cvtph_epi16:** Convert from half-precision to 16-bit integer.
- **_mm_cvtph_epi64:** Convert from half-precision to 64-bit integer.
- **_mm_cvtxph_ps:** Convert from half-precision to FP32.

Note that an extra X appears in some of the intrinsics to differentiate the intrinsics from their older FP16C/Intel AVX-512 F ISA counterparts. Only instructions that could be confused with older instruction sets have an X in their name (for example, the int16 conversion only appears in Intel AVX-512 FP16 so it does not need to be disambiguated).

When an FP16 denormal value is converted to a higher-precision FP32 or FP64 value, the denormal is converted to a normal representation in the output format.

Although the older conversion instructions perform type conversion as expected, they do not support embedded broadcasts. It is recommended to use the newer instructions wherever possible to get some instruction encoding advantages.

Conversions to FP16 format from other data types all take the intrinsic form shown in the following examples:

- **_mm_cvtepi16_ph:** Convert from 16-bit integer to half-precision.
- **_mm_cvtepi64_ph:** Convert from 64-bit integer to half-precision.
- **_mm_cvtxps_ph:** Convert from FP32 to half-precision.

Note that some care must be taken when converting from higher precision types into FP16. For example, conversion from a signed 16-bit integer value to FP16 generates the equivalent integers in FP16, albeit with some small loss possibly (for example, integer values greater than 2048 may be quantized to a nearby integer, not the exact integer). However, a more serious issue is that values that are converted from full-range 16-bit unsigned integer format are converted into FP16 values, which are at the very upper end of the permitted FP16 number range. Almost any numeric operation on such values could lead to overflow and the generation of infinities. In such scenarios, it is beneficial to perform some scaling on the value after conversion, to bring the range of the new values into the middle of the FP16 number range, thereby making it more difficult to hit infinities or denormals through normal compute operations.

Note that some care must be taken when converting to and from integer types to FP16. In particular, it must be noted that not all values in the 16-bit signed integer range of -32768 to +32767 can be represented in FP16. There will be some quantization effects with values above 2048. As discussed in Section 18.5.2, this additional quantizing noise power is negligible in most signal processing applications. However, the 16-bit integer range includes numbers that become close to Inf in FP16 format (values above 65504 are Inf). To avoid potential problems when performing typical signal-processing tasks such as cross-correlations, which operate in volts^2, 16-bit integer values should be scaled after conversion to FP16. A typical scale would be 1/256. In this scheme, 32768 would be converted to 128.00f16. Note that both 128.00^2=16384.00 and $(2/256)^2$=0.00006103515625 are within the normal range of FP16 values. This means that most typical signal-processing operations can be performed with values mostly in the normal range (with $(1/256)^2$ just falling outside of the normal range). So, by the simple expedient of applying a fixed scaling, FP16 representation can be used to comfortably span the dynamic-range presented by 16-bit ADCs and DACs.

18.5.8 APPROXIMATION INSTRUCTIONS AND THEIR USES

In common with Intel AVX-512, the new FP16 instructions support a number of approximation functions, including reciprocal (rcp), and reciprocal-sqrt (rsqrt). Although many instructions in Intel AVX-512 FP16 are accurate to within

0.5 ULP, as guaranteed by IEEE 754, the approximation instructions give very slightly less accurate results, but these are still useful, especially when compared with their equivalents in FP32 and FP64.

In FP32 and FP64 the approximation instructions are quite rough (that is, have a very high ULP error) and can only be used as a substitute for full-precision operations if combined with one or two Newton-Raphson iterations to refine the initial approximation to a point where it becomes sufficiently accurate. However, in FP16 the approximation functions give results that are so close to their full precision results - within 0.5 ULP for 98% of the possible values and within 0.5625 ULP for the remaining 2% of values - that there is no need to add Newton-Raphson iterations. This makes the approximation instructions very useful. They give virtually the correct answer, but with substantial benefits in performance over their full-accuracy counterparts. The following sections examine each approximation instruction in more detail.

18.5.8.1 Approximate Reciprocal

The reciprocal instruction in Intel AVX-512 FP16 behaves almost identically to the equivalent code sequence implemented using a division of the constant 1.0. For example, consider the following two code fragments:

```
__m512h trueRcp = _mm512_div_ph(_mm512_set1_ph(1.0f16), x);      // #1
__m512h approxRcp = _mm512_rcp_ph(x);                            // #2
```

The first, true reciprocal-by-division is guaranteed to be within 0.5 ULP, assuming rounding to nearest-even is used, but it takes approximately 15 cycles in 128 or 256-bit mode, and 24 cycles in 512-bit mode. It has a throughput of one instruction every 8 cycles in 128 or 256-bit mode, or one every 16 cycles in 512-bit mode. In contrast, the approximate reciprocal instruction is within 0.5 ULP for 98% of the possible valid input values, and the remaining 2% of values are within 0.5625 ULP, but it has a latency of only 4 cycles (or 6 cycles in 512-bit mode), and a throughput of 1 cycle (or 1.5 cycles in 512-bit mode). This dramatic improvement in compute performance of rcp_ph for almost no difference in numeric performance makes it ideal whenever that particular use case is required. Only when there is an absolute requirement for IEEE floating-point behavior should the division sequence be used instead.

18.5.8.2 Approximate Division

The two code fragments below show how a division could be implemented:

```
__m512h trueDiv = _mm512_div_ph(lhs, rhs);                       // #1
__m512h approxRcp = _mm512_mul_ph(lhs, mm512_rcp_ph(rhs));       // #2
```

The first of these uses the actual division instruction and is accurate to within 0.5 ULP (that is, correctly rounded, regardless of rounding mode).

The second sequence implements division by multiplying by the reciprocal, where the reciprocal is computed using the approximate function. We have already seen in Section 18.5.8.1 that the reciprocal approximation is very good, and this means that performing division using this sequence also turns out to be very good for most FP16 values. To illustrate how good the approximation of the divide is, consider the heat-map shown in Figure 18-8. It shows how the ULP error changes for all the possible values of divisors and dividends. The green areas show that the approximation sequence gives identical results to a real division operation when the dividend is large and the divisor is small, or when the divisor is large and the dividend is small. The yellow region shows the cases where both the dividend and divisor fall into the middle-range of FP16 values, which is where a numerically well-designed algorithm falls, and indicates that 98% of values are within 0.5625 ULP of being correct, and every possible combination of dividend and divisor is never less accurate than about 1.5 ULP. The only places where the approximate division breaks down is when the divisor is very small (i.e., the left-hand red strip corresponding to the denormals), or very large (that is, the right-hand red strip where the exponent is at, or close to the maximum).[1]

1. For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

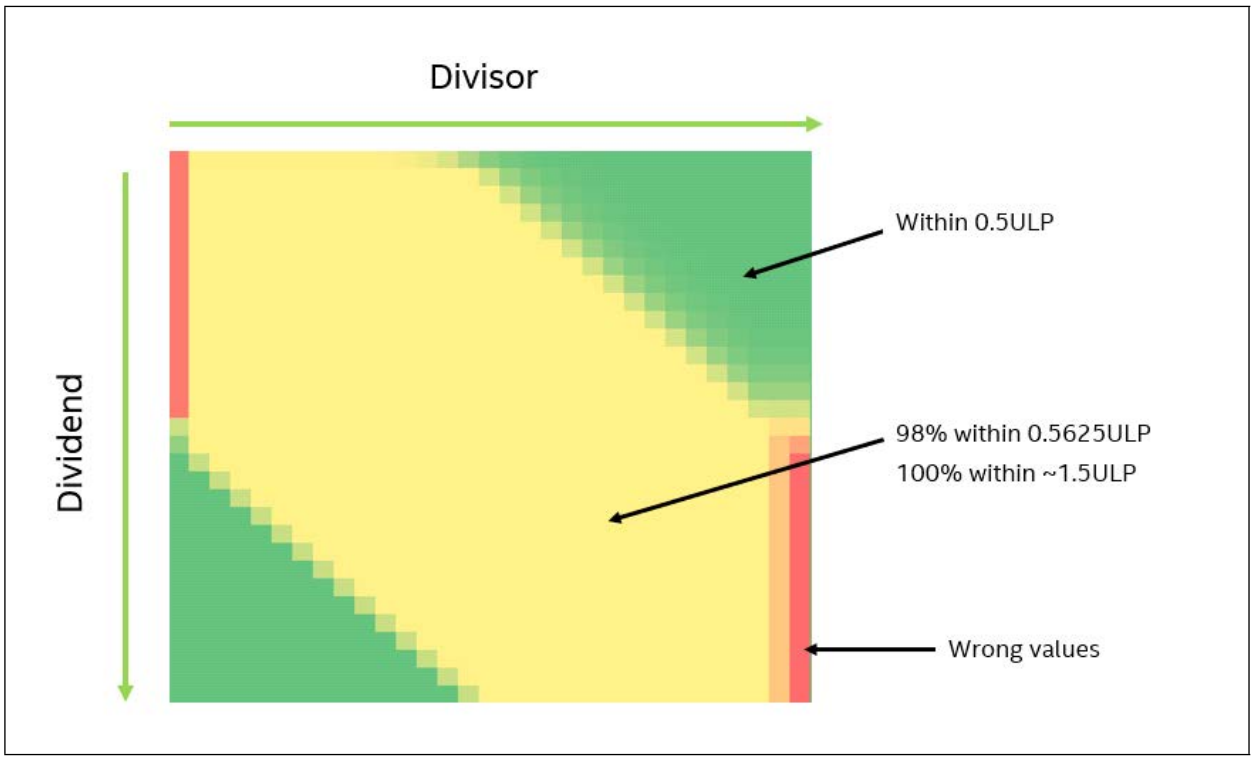


Figure 18-8. Heat-map Showing Relative ULP Error for Different Combinations of Divisor and Dividend Value Ranges

A division instruction is relatively expensive, taking 24 cycles with a throughput of 16 in 512-bit mode. In contrast, both multiply and reciprocal are cheap instructions, even when used in sequence, and consequently the approximation to division is ~3x faster. This speed, coupled with the low error for most FP16 values, means that well-designed algorithms could use the approximation sequence with little disadvantage.

18.5.8.3 Approximate Reciprocal Square Root

The reciprocal square root instruction in Intel AVX-512 FP16 is numerically very good. It gives a value that is within 0.5 ULP for 98% of the valid inputs, and the remaining 2% are within 0.5625 ULP of the true result.

An obvious implementation of a reciprocal square root, which uses the correctly rounded operations, is shown below:

```
__m512h rsqrtSequence = _mm512_div_ph(_mm512_set1_ph(1.0f16), _mm512_sqrt_ph(x));
```

This also gives a very good answer - within 0.5 ULP of the true result for 73% of possible inputs, and within 1 ULP for the remaining 23% - but that is slightly worse than the **_mm_rsqrt_ph** instruction itself, so the approximation intrinsic should be used in preference.

The approximate reciprocal square root takes 6 cycles, compared to the alternative sequence above that takes 48 cycles when in 512-bit mode.

18.5.9 APPROXIMATE SQUARE ROOT

Although a square root instruction exists and is within 0.5 ULP of the true answer, it is possible to combine a product and rsqrt to get a good approximation, as shown in the code fragment below:

```
__m512h sqrtSequence = _mm512_mul_ph(x, _mm512_rsqrt_ph(x));
```

This sequence gives an answer that is identical to that from the sqrt instruction for 70% of the possible input values and is never more than 1 ULP away from the true result for any FP16 input value. The throughput of the product is twice the throughput of the reciprocal square root approximation, allowing for some flexibility in internal scheduling. The speed of this instruction sequence, coupled with its negligible error, makes it suitable for fast sqrt for any algorithms except those that require guaranteed IEEE rounding.

18.6 USING EXISTING INTEL® AVX-512 INSTRUCTIONS TO AUGMENT FP16 SUPPORT

Intel AVX-512 FP16 provides purely numeric operations that require hardware support and cannot easily be implemented in any other way. For all other related operations needed to support the use of vector FP16 (for example, permuting FP16 elements in a register), it is necessary to use the instructions that are already provided as part of the existing Intel AVX-512 instruction set. As a convenience, the compiler provides many support functions. For example, the intrinsic called _mm512_mask_blend_ph, which blends FP16 elements from two registers into one, is implemented using the underlying mask_blend_epi16 instruction. In cases where the compiler does not provide such convenience functions it will be necessary for programmers to create a wrapper to handle this themselves. In this section we show how such functions could be implemented, list some of the common convenience instructions, and show one example where extra performance can be achieved by exploiting the Intel AVX-512 instruction set to handle floating-point comparisons more efficiently.

18.6.1 USING EXISTING INSTRUCTIONS TO EXTEND INTEL® AVX-512 FP16 INTRINSICS

Suppose we wish to use a bit mask to compress the elements of an FP16 vector register, creating something that would act as you would expect from a non-existent intrinsic called **_mm512_mask_compress_ph.** Although such an intrinsic does not exist, it can be created as shown in Example 18-4.

Example 18-4. Function to Implement the 16-Bit Compress Operation on FP16 Vector Elements

```
__m512h compress_ph(__m512h src, __mmask32 mask, __m512h value)
{
    const auto asInt16 = _mm512_castph_si512(value);
    const auto src16 = _mm512_castph_si512(src);
    const auto comp = _mm512_mask_compress_epi16(src16, mask, asInt16);
    return _mm512_castsi512_ph(comp);
}
```

The strategy followed in the example is to take the incoming vector of FP16 values and recast to a vector of **int16_t** values using the castph_si512 intrinsic. The newly cast values are then processed as though they are a vector of int16_t elements instead. This step does not change the individual 16-bit element blocks; it just moves them around within the register. On completion, the value is recast back to its original type as a vector of FP16 elements.

Note that the cast operations have no runtime impact and are purely used to inform the compiler that the programmer is treating the underlying bits in the incoming register as though they were a different type. No type

conversion takes place. In practice, the entire code sequence in the example function collapses into a single compress instruction.

This same strategy can be used to apply any sort of data movement instructions as a method of moving data around within FP16 vector registers. The code is somewhat verbose but can be easily hidden away as a library function. Furthermore, many common utility intrinsics of this sort have already been implemented in the Intel OneAPI compiler and can be used directly. It should only be necessary to build additional intrinsic support functions for more unusual operations.

In addition to data movement instructions, other bitwise operations like abs, nabs, negate, copy-sign, and so on, can also be implemented using the underlying Intel AVX-512 foundation instructions.

18.6.2 COMMON CONVENIENCE INTRINSICS

Convenience instructions are provided for the common cases where FP16 support is implemented with existing Intel AVX-512 instructions, without requiring the definition of verbose intrinsics given in the compress_ph example above. Table 18-6 provides a list of some common convenience intrinsics.

Table 18-6. Conjugation Instructions

Mode	Purpose and Implementation
_mm512_conj_pch	Compute the conjugate of a complex number by using bitwise XOR operation to flip the sign bit of the imaginary elements.
_mm512_abs_ph	Compute the absolute numeric value of an FP16 element by using a bitwise AND instruction to mask off the sign bit.
_mm512_mask_blend_ph	Use the underlying _mm512_mask_blend_epi16 intrinsic to provide an FP16 (_ph) equivalent.
_mm512_permute[x]var_ph	Reorder FP16 elements from one or two source registers.
_mm512_reduce_[add/min/max]_ph	Generate a sequence of instructions that performs a reduction operation across all the elements of an FP16 vector register. This is more complicated than the other examples because it performs a sequence of permutes and reorders to pull the data together and intersperses those operations with numeric reduction operations that perform addition, multiplication, minimum, and so on.

18.6.3 USING INTEGER COMPARISONS FOR FAST FLOATING-POINT COMPARISON

IEEE floating-point values have the property that all non-NaN values that are either both known to be positive or known to have different signs can be directly compared by treating their bit pattern as a 16-bit signed integer. This

does not work when both values are negative. The fast-integer property can be exploited to give a low-latency minimum (or maximum) function as shown in the code fragment in Example 18-5.

Example 18-5. Function that Performs Fast Floating-Point Minimum Using Integer Instructions

```
// Assume the inputs are sane values, and either both positive or opposite signs.
__m128h fast_special_min(__m128h lhs, __m128h rhs)
{
    const auto lhsInt16 = _mm_castph_si128(lhs);
    const auto rhsInt16 = _mm_castph_si128(rhs);
    const auto smallest = _mm_min_epi16(lhsInt16, rhsInt16);
    return _mm_castsi128_ph(smallest);
}
```

By using the int16_t minimum instead, the instruction takes only 1 cycle to execute, which is faster than the equivalent FP16 minimum, and can be used to accelerate latency or dependency-sensitive code. Note however that the throughput is lower than the equivalent FP16 minimum instruction, so code that is exclusively performing minimum operations may do better using the FP16 minimum.[1]

For comparison operations all data types take the same number of cycles to compare so using the equivalent int16_t form of the instruction to perform a comparison makes no difference to performance.

18.7 MATH LIBRARY SUPPORT

The math libraries provided with Intel compilers offer full functionality for the float16 data type (_Float16). Compiler support for the float16 data type can be enabled with ***-arch=sapphirerapids*** (Ice Lake Server compiler). Float16-specific optimizations are available for vectorized math library calls.

Scalar math library functionality is available in the LIBM. These functions have not yet been optimized for Intel AVX-512 FP16 and currently rely on existing float32 implementations. Scalar float16 function names use the f16 suffix (for example, expf16, logf16, sinf16, cosf16).

At the default accuracy level (4 ULP or better), most common functions in the short vector math library (SVML) are optimized to take full advantage of the new Intel AVX-512 FP16 instruction set. Higher accuracy versions (1 ULP) are also available, and most have been optimized; however, the 1 ULP versions frequently rely on single precision computation to achieve the required accuracy. It is thus expected that the 4 ULP implementations will provide noticeably better performance. SVML calls are generated by the compiler as part of loop vectorization. Compiler intrinsics for SIMD float16 calls (for example, _mm_log_ph) are not yet available.

The Intel AVX-512 FP16 instruction set includes several operations that support efficient implementation of math libraries. These operations are extensions of Intel AVX-512 transcendental support instructions and include VGETEXP, VGETMANT, VSCALEF, VFPCLASS, VREDUCE, VRNDSCALE (in addition to VRCP, VRSQRT).

VGETEXP (get normalized exponent) and **VGETMANT** (get normalized mantissa) are used together in implementations of functions such as log(), pow(), cbrt(). In the absence of these operations, denormal and special inputs would require treatment in a separate path (or alternatively, a slower main path that treats all inputs correctly). Denormals are reasonably likely to occur as inputs to float16 SVML calls due to the narrow float16 format range. The relative frequency of special inputs also increases with a wider SIMD length (32 packed float16 inputs per 512-bit SIMD register), so it is especially helpful to avoid branches. As an example, VGETEXP and VGETMANT can be used to reduce the log() computation to log(x)=VGETEXP(x)*log(2) + log(VGETMANT(x,8)). VGETMANT with an

1. For workloads and configurations visit www.Intel.com/PerformanceIndex. Results may vary.

immediate value of 8 returns the normalized mantissa (in the [1,2) range) for positive inputs, and QNaN_Indefinite for negative inputs (which helps with special case handling).

VSCALEF(a,b)=a*$2^{floor(b)}$ is used in exponential and power functions; other possible applications include software division. This operation helps with correct overflow and underflow treatment in the main path. It also includes support for special exp() cases, thus eliminating the need for branches or other fixup code for this function family.

VFPCLASS is used to test for multiple special case categories (sNaN, negative finite, denormal, -Infinity, +Infinity, -0, +0, qNAN). This helps when redirecting special inputs to a secondary path (for example, in the pow() function), or to generate a fixup mask for setting special case results in the main path.

VRNDSCALE (round to specified number of fractional bits, using specified rounding mode) is used in function argument reduction, and to help generate lookup table indices (also as part of argument reduction). VRNDSCALE is a generalized form of round-to-integral, so it provides ceil/floor/trunc functionality, and also helps with floating-point remainder operations.

VREDUCE is closely related to VRNDSCALE: VREDUCE(x, imm) = x - VRNDSCALE(x,imm). This instruction helps further speed up argument reduction for certain functions (for example, exp2, pow).

The existing Intel AVX-512 permute operations (VPERM, VPERMT2, VPERMI2 for 16-bit and 32-bit data) provide fast vector gather support for those implementations that need lookup tables (up to 32 16-bit entries for VPERMW, up to 64 16-bit entries for VPERMT2W/VPERMI2Wn operations).

CHAPTER 19
CRYPTOGRAPHY & FINITE FIELD ARITHMETIC ENHANCEMENTS

Several instruction extensions designated for acceleration of cryptography flows and finite field arithmetic are available, beginning with the Ice Lake Client microarchitecture. The ISA extensions include Vector AES, VPCLMULQDQ, Galois Field New Instructions (GFNI), and AVX512_IMFA, also known as VPMADD52. The following sections describe these extensions and provide examples and simple comparisons to previous implementation code.

See the Intel® 64 and IA-32 Architectures Software Developer's Manual for the complete instruction definitions.

Intel implements support for the most common cryptography algorithms, supporting main public libraries and commonly used software. The following sections describe the instructions briefly.

19.1 VECTOR AES

The Vector AES extension supports the vectorization of the previously announced Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI)[1] . The new instructions support parallel execution for up to four blocks of input within a single instruction. The extended ISA, namely VAESENC, VAESENCLAST, VAESDEC, and VAESDECLAST, are intended for performance acceleration of the relevant AES mode of operations and multi-buffer implementations. The new instructions accelerate AES modes by up to 3.3x compared to previous code supported by Intel AES-NI.

Below is a snippet of AES-ECB mode of operation code, implemented on AT-T Assembly, emphasizing the legacy Intel AES-NI vs. Vector AES.

Example 19-1. Legacy Intel® AES-NI vs. Vector AES

Legacy Intel® AES-NI - AES ECB Encryption	Vector AES - AES ECB Encryption

// rcx = pointer to Key Expansion
movdqa 16*1(%rcx), %xmm9
// xmm1 - xmm8 - 8 blocks of AES
// 1st round of AES for 8 blocks
aesenc %xmm9, %xmm1
aesenc %xmm9, %xmm2
aesenc %xmm9, %xmm3
aesenc %xmm9, %xmm4
aesenc %xmm9, %xmm5
aesenc %xmm9, %xmm6
aesenc %xmm9, %xmm7
aesenc %xmm9, %xmm8
movdqa 16*2(%rcx), %xmm9
// 2nd Round of AES for 8 Blocks
aesenc %xmm9, %xmm1
aesenc %xmm9, %xmm2
	// rcx = pointer to Key Expansion
	// broadcasting key to zmm0
	vbroadcasti64x2 1*16(%rcx), %zmm0
	// 1st round of AES for 8 blocks
	vaesenc %zmm0, %zmm1, %zmm1
	vaesenc %zmm0, %zmm2, ,%zmm2
	vbroadcasti64x2 2*16(%rcx), %zmm0
	// 2nd Round of AES for 8 Blocks
	vaesenc %zmm0, %zmm1, %zmm1
	vaesenc %zmm0, %zmm2, %zmm2

1. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

Example 19-1. Legacy Intel® AES-NI vs. Vector AES (Contd.)

Legacy Intel® AES-NI - AES ECB Encryption	Vector AES - AES ECB Encryption
aesenc %xmm9, %xmm3 aesenc %xmm9, %xmm4 aesenc %xmm9, %xmm5 aesenc %xmm9, %xmm6 aesenc %xmm9, %xmm7 aesenc %xmm9, %xmm8	
Baseline: 1x	Speedup: 3.3x

The code above demonstrates the ability of implementing AES in ECB mode of operation, using 8 parallel buffers, implemented on legacy vs. Vector AES. The same acceleration can be applied to other modes of operation, such as AES-CTR and AES-CBC, and also to more elaborate schemes such as AES-GCM. The latter one requires fast computations of a hash function, namely GHASH, which can be accelerated using the VPCLMULQDQ new instruction.

19.2 VPCLMULQDQ

Carry-less multiplication, namely PCLMULQDQ, was previously introduced on the Intel® Core™ processor family based on Westmere microarchitecture[1]. In newer architectures, beginning with Ice Lake Client microarchitecture, Intel introduces the vectorization of PCLMULQDQ, namely VPCLMULQDQ, supporting acceleration of up to 4x compared to its legacy. The new instruction is used for polynomial multiplication over binary fields used on current cryptography algorithms such as AES-GCM. The new instruction may also be useful for the upcoming Post-Quantum Cryptography project submissions, used for BIKE and others. Such usages emphasizes the importance of the current use of VPCLMULQDQ. A common use case for using the instruction can be seen on GHASH computation, with four different carry-less multiplications done within a single instruction, using the wide 512-bit registers. This use case elaborates the performance of AES-GCM, which is the main mode of operation used on AES.

19.3 GALOIS FIELD NEW INSTRUCTIONS

Galois Field new instructions are recently introduced in newer architecture, beginning with Ice Lake Client microarchitecture. The new instructions, namely VGF2P8MULB, VGF2P8AFFINEQB, and VGF2P8AFFINEINVQB, allow software flows to perform vector and matrix multiplications over GF(2^8) on the Intel AVX512 architectural registers. The wide usages of these instructions vary from Reed-Solomon code implementation, to different encryption schemes such as the Chinese encryption scheme - SM4 and others.

1. https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode

Example 19-2. SM4 GFNI Encryption Round Example

```
LAFFINE_IN:
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34
.byte 0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34,0x52,0xBC,0x2D,0x02,0x9E,0x25,0xAC,0x34

.LAFFINE_OUT:
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.byte 0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7,0x19,0x8b,0x6c,0x1e,0x51,0x8e,0x2d,0xd7
.globl SM4_ENC_ECB_AVX512_GFNI
SM4_ENC_ECB_AVX512_GFNI:
vmovdqa64   .LAFFINE_IN(%rip), %zmm10
   vmovdqa64   .LAFFINE_OUT(%rip), %zmm11
   …
   /* Load data swapped LE-BE in transposed way - each block's double word is found on
     different AVX512 register
   */

.Rounds:
   // Initial xor between the 2nd, 3rd, 4th double word to key
   vpbroadcastd 4*0(key), %zmm6
   vpternlogd $0x96, %zmm1, %zmm2, %zmm3
   vpxorq %zmm3, %zmm6, %zmm6
   /* Sbox phase */
   vgf2p8affineqb $0x65, %zmm10, %zmm6, %zmm6
   vgf2p8affineinvqb $0xd3, %zmm11, %zmm6, %zmm6
   /* Done Sbox , Linear rotations start xor with 1st double word input*/
   vprold $2, %zmm6, %zmm12
   vprold $10, %zmm6, %zmm13
   vprold $18, %zmm6, %zmm7
   vprold $24, %zmm6, %zmm14
   vpternlogd $0x96, %zmm6, %zmm12, %zmm0
   vpternlogd $0x96, %zmm13, %zmm7, %zmm14
   vpxord %zmm14, %zmm0, %zmm0
   /* Linear part done - round complete */
```

19.4 INTEGER FUSED MULTIPLY ACCUMULATE OPERATIONS (AVX512_IFMA - VPMADD52)

Beginning with Ice Lake Client microarchitecture, Intel introduces AVX512_IFMA, namely VPMADD52. The new instructions, VPMADD52LUQ and VPMADD52HUQ, multiply 8x52-bit unsigned integers found in the 512-bit wide registers, produce the high and low halves of the result, and add to the 64-bit accumulators. The instructions are designated for big number multiplication, assuming the inputs are using radix 252. The new instructions can be used for accelerating modular exponent computation code, which is widely used on RSA. Code usages can be already seen on OpenSSL[1].

1. https://www.openssl.org/

CHAPTER 20
INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

This chapter aims to help low-level DL programmers optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP and Emerald Rapids microarchitectures. It extends the public documentation on optimizing DL code with DL Boost instructions in Section 20.8.

It explains how to detect processor support in Intel® Advanced Matrix Extensions (Intel® AMX) architecture (Section 20.1). It provides an overview of the architecture (Section 20.2) and presents Intel AMX instruction throughput and latency (Section 20.3). It also discusses software optimization opportunities for Intel AMX (Section 20.5 through Section 20.18), TILECONFIG/TILERELEASE and compiler ABI (Section 20.19), Intel AMX state management and system software aspects (Section 20.20), and the use of Intel AMX for higher precision GEMMs (Section 20.21).

Table 20-1. (Sheet 1 of 2)Intel® AMX-Related Links

Description	Description
Intel® AMX architecture definitions in the Intel® 64 and IA-32 Architecture Software Developer's Manual	Describes the architecture and programming environment of the Intel® 64 and IA-32 architectures.
Buildable and executable templates of code examples for this chapter.	Contains the source code examples described in the "Intel® 64 and IA-32 Architectures Optimization Reference Manual."
Open VINO™ Documentation	Provides reference documents that guide you through the OpenVINO toolkit workflow, from preparing and optimizing models to deploying them in deep learning (DL) applications.
oneDNN Documentation	oneAPI Deep Neural Network Library Developer Guide and Reference.
oneDNN GitHub	oneAPI Deep Neural Network Library (oneDNN).
Intel® Optimization TensorFlow Installation Guide	Technical documentation and guide to the optimization of Intel® Optimization for TensorFlow*.
PyTorch Documentation	Library and resources related to PyTorch.
PyTorch GitHub	Buildable and executables related to tensors and dynamic neural networks in Python with strong GPU acceleration.
Intel® Neural Compressor (INC) GitHub	Provides unified APIs for SOTA model compression techniques, such as low precision (INT8/INT4/FP4/NF4) quantization, sparsity, pruning, and knowledge distillation on mainstream AI frameworks such as TensorFlow, PyTorch, and ONNX Runtime.
Tips for measuring the performance of matrix multiplication using Intel® MKL	Provides tips on how to measure the single-precision general matrix multiply (SGEMM) function performance on Intel® Xeon® processors.
GitHub Repository	Code samples related to Intel® AMX.

Table 20-1. (Contd.)(Sheet 2 of 2)Intel® AMX-Related Links

Description	Description
Using XSTATE features in user space applications	Linus Torvalds' GitHub discussion of XState.
GetThreadEnabledXStateFeatures function (winbase.h)	Microsoft's discussion of this function.
UpdateProcThreadAttribute function (processthread-sapi.h)	Microsoft's discussion of this function.

20.1 DETECTING INTEL® AMX SUPPORT

Use the CPUID instruction described in Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual to determine whether the processor you are executing on supports Intel AMX at the hardware level.

Specifically, when issuing the CPUID instruction with the EAX register set to 7 and the ECX register set to 0, the instruction returns in the EDX register an indication of Intel AMX support of bits 22, 24, and 25.

- They are all set to 0 if Intel AMX is not supported.
- They are all set to 1 if Intel AMX is supported.

The next step is to check whether the OS has enabled the Intel AMX state:

1. Issue the CPUID instruction again to check whether the OS supports the XGETBV instruction,
2. Use the instruction to check whether the OS has enabled the Intel AMX state save/restore.

When issuing the CPUID instruction with the EAX register set to 1, the instruction returns an indication of XGETBV support in bit 26 of the ECX register. If bit 26 is set, when issuing the XGETBV instruction with ECX register set to 0, the instruction returns an indication of OS support in saving and restoring the Intel AMX state in bits 17 and 18 of the EAX register. Both bits should be set to use the Intel AMX instructions. For additional CPUID information about Intel AMX, see Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual

Operating systems may require calling an OS API to allocate Intel AMX state. Visit Linux API and Windows APIs for more detailed information. Please see Section 20.20 for more information about Intel AMX state management.

20.2 INTEL® AMX MICROARCHITECTURE OVERVIEW

General Intel AMX microarchitecture overview is available in Chapter 18 of Volume 1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual.

20.2.1 INTEL® AMX FREQUENCIES

See Section 2.6.3 for a discussion about the connection between max frequency, frequency license, and Instruction Set Architecture covering Intel AVX technologies up to Intel® AVX-512 Instruction Set. Intel AMX adds yet another license level whose max frequency is usually lower than that of the Intel AVX-512 license.

When the Intel AMX unit utilization is lower than 15%, the processor may exceed the nominal max frequency associated with the Intel AMX license.

20.3 INTEL® AMX INSTRUCTIONS THROUGHPUT AND LATENCY

Several Intel AMX instructions are available. Two instructions (TileLoad*) load data from the memory hierarchy into the tile registers, and one instruction (TILESTORE) stores the contents of a tile register into the DCU (Data Cache Unit– first level cache). Other instructions (TDP*) execute the matrix multiplication, operating on two input tile registers and writing the result into a third tile register. Additionally, there are some less-frequently used instructions. The following table provides the instruction throughput and latency counted in cycles.

Table 20-2. Intel® AMX Instruction Throughput and Latency

Instruction	Throughput	Latency
LDTILECFG	Not Relevant	204
STTILECFG	Not Relevant	19
TILETRELEASE	Not Relevant	13
TDP/*	16	52
TILELOADD	8	45
TILELOADDT1	33	48
TILESTORED	16	Not Relevant
TILEZERO	0	16

NOTE

Due to the high latency of the LDTILECFG instruction, we recommend issuing a single pair of LDTILECFG and TILERELEASE operations per Intel AMX-based DL layer implementation.

20.4 DATA STRUCTURE ALIGNMENT

The GEMM and Convolution input/output data structures must be 64-byte aligned for optimal performance but should not be aligned to 128-byte, 256-byte, etc. For more details, see Tip 6 in *Tips for Measuring the Performance of Matrix Multiplication Using Intel® MKL*.

20.5 GEMMS / CONVOLUTIONS

20.5.1 NOTATION

The following notation is used for the matrices (A, B, C) and the dimensions (M, K, N) in matrix multiplication (GEMM).

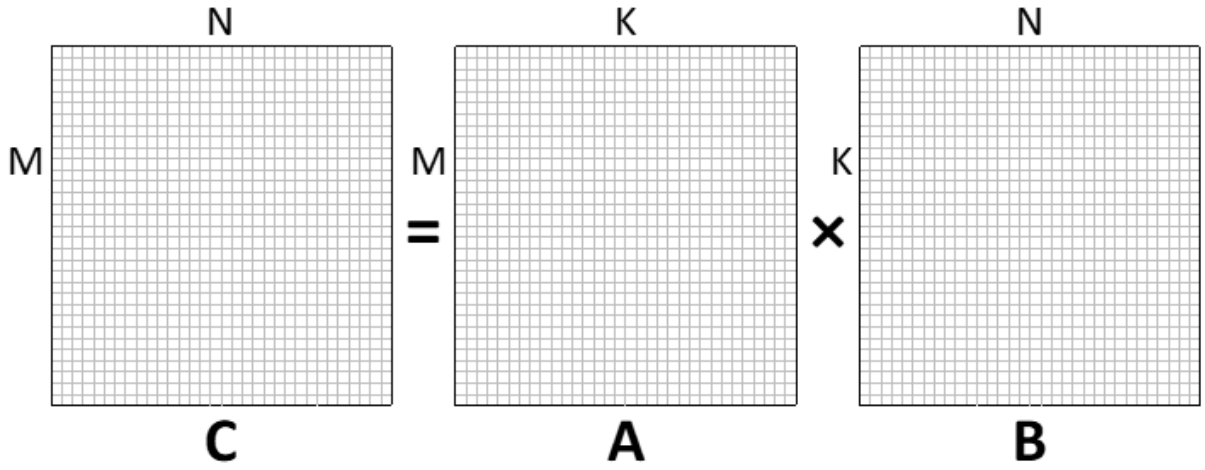


Figure 20-1. Matrix Notation

20.5.2 TILES IN THE INTEL® AMX ARCHITECTURE

The Intel AMX instruction set operates on tiles: large two-dimensional registers with configurable dimensions. The configuration is dependent on the type of tile.

- A-tiles can have between 1-16 rows and 1-MAX_TILE_K columns.
- B-tiles can have between 1-MAX_TILE_K rows and 1–16 columns.
- C-tiles can have between 1-16 rows and 1–16 columns.

MAX_TILE_K=64/sizeof(type_t), and type_t is the data type being operated on. Therefore, MAX_TILE_K=64 for (u)int8 data, and MAX_TILE_K=32 for bfloat16 data. The dimensions here are mathematical/logical. For more detail about mapping to tile register configuration parameters, see the Intel® Architecture Instruction Set Extensions Programming Reference.

The data type residing in the tiles also varies depending on the type of tile. A-tiles and B-tiles contain data of **type_t**, which can be (u)int8 or bfloat16.

C-tiles contain data of type res_type_t:

- int32 if type_t=(u)int8
- float if type_t=bfloat16

Thus, a maximum-sized tile multiplication operation for (u)int8 data type looks this way:

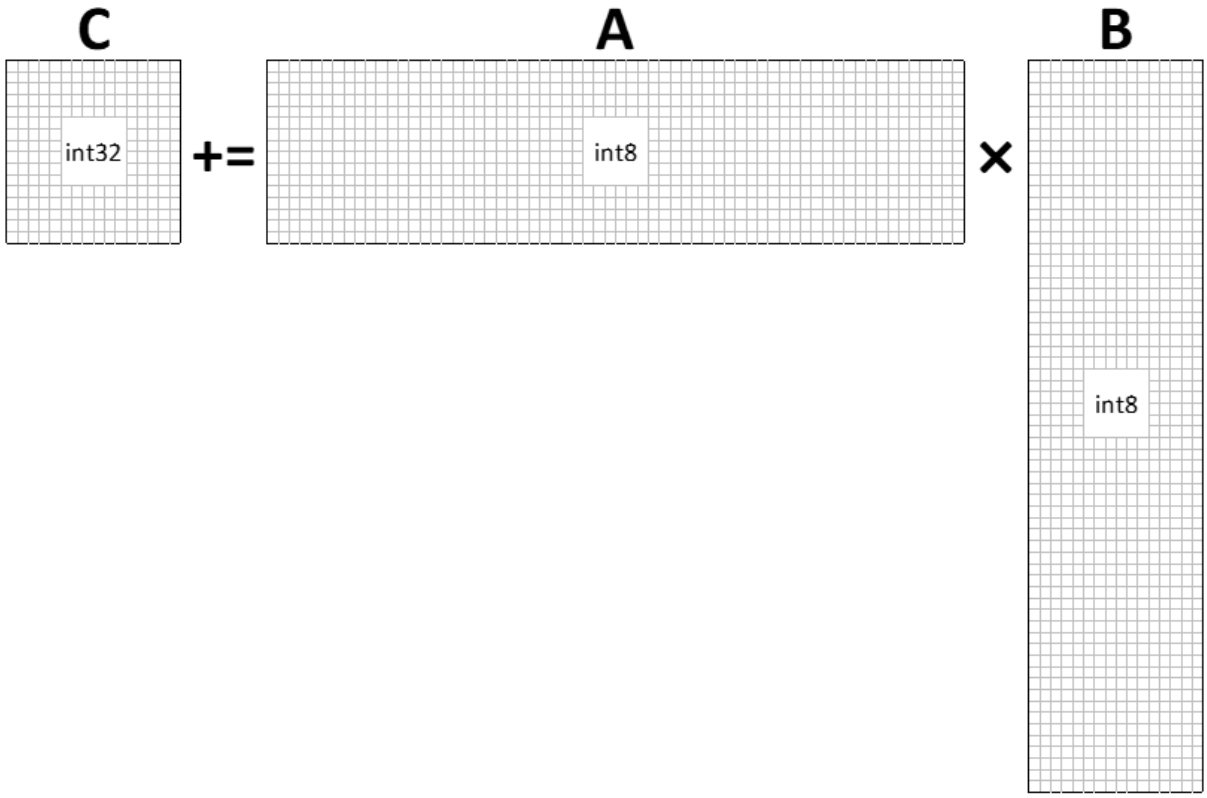


Figure 20-2. Intel® AMX Multiplication with Max-sized int8 Tiles

TileLoad and TILESTORE Instructions

The tiles are loaded from memory with the TileLoad instruction and stored to memory with a TILESTORE instruction. The TILELOAD/TILESTORE instructions receive the following parameters:

- The destination/source tile of the TILELOAD/TILESTORE.
- The source/destination location in memory for the TILELOAD/TILESTORE.
- The stride (bytes) in memory between subsequent rows of the tile.

Lines 6—10 in Example 20-1 illustrate how a tile is loaded from memory.

Example 20-1. Pseudo-Code for the TILEZERO, TILELOAD, and TILESTORE Instructions

```
template<size_t rows, size_t bytes_cols> class tile {
public:
  friend void TILEZERO(tile& t) {
    memset(t.v, 0, sizeof(v));
  }
  friend void TILELOAD(tile& t, void* src, size_t bytes_stride) {
    for (size_t row = 0; row < rows; ++row)
      for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
        t.v[row][bcol] = static_cast<int8_t*>(src)[row*bytes_stride + bcol];
  }
friend void TILESTORE(tile& t, void* dst, size_t bytes_stride) {
    for (size_t row = 0; row < rows; ++row)
      for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
        static_cast<int8_t*>(dst)[row*bytes_stride + bcol] = t.v[row][bcol];
  }
template <class TC, class TA, class TB>
friend void tdp(TC &tC, TA &tA, TB &tB);
private:
  int8_t v[rows][bytes_cols];
};

// clang-format on

template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB)
}
```

For the sake of readability, a tile template class abstraction is introduced. The number of rows in the tile and the number of column bytes per row parametrizes the abstraction.

20.5.3 B MATRIX LAYOUT

Like the Intel® DL Boost use case, the B matrix must undergo a re-layout before it can be used within the corresponding Intel AMX multiply instruction. The re-layout procedure is as follows:

Example 20-2. B Matrix Re-Layout Procedure

Example 1-3

```
#define KPACK (4/sizeof(type_t))// Vertical K packing into Dword

type_t B_mem_orig[K][N];// Original B matrix
type_t B_mem[K/KPACK][N][KPACK];// Re-laid B matrix

for (int k = 0; k < K; ++k)
  for (int n = 0; n < N; ++n)
    B_mem[k/KPACK][n][k%KPACK] = B_mem_orig[k][n];
```

The following tables illustrate the data re-layout process for a 64x16 int8 B matrix and a 32x16 bfloat16 B matrix (corresponding to the maximum-sized B-tile):

Example 20-3. Original Layout of 32x16 bfloat16 B-Matrix

Example 1-4

```
Row0: {0, 1, 2, 3, … 14, 15}
Row1: {16, 17, 18, 19, … 30, 31}
Row2: {32, 33, 34, 35, … 46, 47}
Row3: {48, 49, 50, 51, … 62, 63}
Row4: {64, 65, 66, 67, … 68, 79}
Row5: {80, 81, 82, 83, … 94, 95}
Row6: {96, 97, 98, 99, … 110, 111}
Row7: {112, 113, 114, 115, … 128, 127}
Row8: {128, 129, 130, 131, … 142, 143}
Row9: {144, 145, 146, 147, … 158, 159}
…
Row30: {480, 481, 482, 483, … 494, 495}
Row31: {496, 497, 498, 499, … 510, 511}
```

Example 20-4. Re-Layout of 32x16 bfloat16 B-Matrix

Example 1-5

```
Row0: {0, 16, 1, 17, 2, 18, … 14, 30, 15, 31}
Row1: {32, 48, 33, 49, 34, 50, … 46, 62, 47, 63}
Row2: {64, 80, 65, 81, 66, 82, … 78, 94, 79, 95}
Row3: {96, 112, 97, 113, 98, 114… 110, 126, 111, 127}
Row4: {128, 114, 129, 115,130, 116, … 142, 158, 143, 159}
Row5: {160, 176, 161, 177, 162, 178, … 174, 190, 175, 191}
Row6: {192, 208, 193, 209, 194, 210, … 206, 222, 207, 223}
Row7: {224, 240, 225, 245,226, 246, … 238, 254, 239, 255}
Row8: {256, 272, 257, 273, 258, 274, … 270, 286, 271, 287}
Row9: {288, 304, 289, 305, 290, 290, … 302, 318, 303, 319}
…
Row14: {448, 464, 449, 465, 450, 466,… 462, 478, 463, 479}
Row15: {480, 496, 481, 497, 482, 498,… 494, 510 495, 511}
```

Example 20-5. Original Layout of 64 x 16 unt8 B-Matrix

Example 1-6

```
Row0: {0, 1, 2, 3, … 14, 15}
Row1: {16, 17, 18, 19, … 30, 31}
Row2: {32, 33, 34, 35, … 46, 47}
Row3: {48, 49, 50, 51, … 62, 63}
Row4: {64, 65, 66, 67, … 68, 79}
Row5: {80, 81, 82, 83, … 94, 95}
Row6: {96, 97, 98, 99, … 110, 111}
Row7: {112, 113, 114, 115, … 128, 127}
Row8: {128, 129, 130, 131, … 142, 143}
Row9: {144, 145, 146, 147, … 158, 159}
…
Row62: {992, 993, 994, 995, … 1006, 1007}
Row63: {1008, 1009, 1010, 1011,… 1022 1023}
```

Example 20-6. Re-Layout of 64x16 bfloat16 B-Matrix

Example 1-7

```
Row0: {0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, … 15, 31, 47, 63}
Row1: {64, 80, 96, 112, 65, 81, 97, 113, … 79, 95, 111, 127}
Row2: {128, 144, 160, 176, 129, 145, … 143, 159, 175, 191}
Row3: {192, 208, 224, 240, 193, … 207, 223, 239, 255}
Row4: {256, 272, 288, 304, 257, … 271, 287, 303, 319}
Row5: {320, 336, 352, 368, 321, … 335, 351, 367, 383}
Row6: {384, 400, 416, 432,385, … 399, 415, 431, 447}
Row7: {448, 464, 480, 496, 449, … 463, 479, 495, 511}
Row8: {512, 528, 544, 560,513 … 527, 543, 559, 575}
Row9: {576, 592, 608, 624, 577… 591, 607, 623, 639}
…
Row14: {896 912, 928, 944, 897,… 911, 927, 943, 959}
Row15: {960, 976, 992, 1008, 961, … 975, 991, 1007, 1023}
```

20.5.4 STRAIGHTFORWARD GEMM IMPLEMENTATION

This is GEMM reference code. Its performance is sub-optimal. Please refer to Section 20.5.5.3 for optimal GEMM code. Begin implementation by defining the following:

Example 20-7. Common Defines

```
#define M ...                          // Number of rows in the A or C matrices
#define K ...                          // Number of columns in the A or rows in the B matrices
#define N ...                          // Number of columns in the B or C matrices
#define M_ACC ...                      // Number of C accumulators spanning the M dimension
#define N_ACC ...                      // Number of C accumulators spanning the N dimension
#define TILE_M ...                     // Number of rows in an A or C tile
#define TILE_K ...                     // Number of columns in an A tile or rows in a B tile
#define TILE_N ...                     // Number of columns in a B or C tile

typedef ... type_t;                    // The type of data being operated on
typedef ... res_type_t;                // The data type of the result

#define KPACK (4/sizeof(type_t))       // Vertical K packing into Dword

type_t A_mem[M][K];                    // A matrix
type_t B_mem[K/KPACK][N][KPACK];       // B matrix
res_type_t C_mem[M][N];                // C matrix

template<size_t rows, size_t bytes_cols> class tile;
template<class T> void TILEZERO (T& t);
template<class T> void TILELOAD (T& t, void* src, size_t stride);
template<class T> void TILESTORE(T& t, void* dst, size_t stride);
template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB) {
   int32_t v;
   for (size_t m = 0; m < TILE_M; m++) {
      for (size_t k = 0; k < TILE_K / KPACK; k++) {
         for (size_t n = 0; n < TILE_N; n++) {
            memcpy(&v, &tC.v[m][n * 4], sizeof(v));
            v += tA.v[m][k * 4] * tB.v[k][n * 4];
            v += tA.v[m][k * 4 + 1] * tB.v[k][n * 4 + 1];
            v += tA.v[m][k * 4 + 2] * tB.v[k][n * 4 + 2];
            v += tA.v[m][k * 4 + 3] * tB.v[k][n * 4 + 3];
            memcpy(&tC.v[m][n * 4], &v, sizeof(v));
         }
      }
   }
}
```

Data type_t is the type being operated upon, i.e., signed/unsigned int8 or bfloat16. For the description of KPACK, see Section 20.5.5. The tile template class and the three functions that operate on it are the same as the ones introduced in Example 20-7. TILEZERO (t) resets the contents of tile t to 0, TILELOAD(t, src, stride) and loads tile t with the contents of data at src with a stride of stride between consecutive rows. TILESTORE(t, dst, stride) stores the contents of tile t to dst with a stride of stride between consecutive rows. TDP(tC,tA,tB) also performs a matrix multiplication

equivalent of tC=tC+tA×tB. In reality, tiles are defined by known compile-time integers, and the actual code operating on tiles looks slightly different. Please visit the GitHub Repository for proper usage.

Example 20-8 is a simple implementation of the GEMM of the matrices stored in A_mem and B_mem.

Example 20-8. Reference GEMM Implementation

```
for (int n = 0; n < N; n += N_ACC*TILE_N) {
 for (int m = 0; m < M; m += M_ACC*TILE_M) {
  tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
  for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
   for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
    TILEZERO(tC[m_acc][n_acc]);

  for (int k = 0; k < K; k += TILE_K) {
   for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
    tile<TILE_K/KPACK, TILE_N*KPACK> tB;
    TILELOAD(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
    for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
     tile<TILE_M, TILE_K*sizeof(type_t)> tA;
     TILELOAD(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
     tdp(tC[m_acc][n_acc], tA, tB);
    }
   }
  }
  for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
   for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
    TILESTORE(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
   }
  }
 }
}
```

20.5.5 OPTIMIZATIONS

20.5.5.1 Minimizing Tile Loads

Redundant tile loads may severely impact performance due to the sizable amount of data loaded into the tiles, unnecessary cache evictions, etc. To minimize tile loads, it is essential to utilize the data as completely as possible once loaded into the tile.

Location of the K Loop: Outside of the M_ACC and N_ACC Loops

The three loops in lines 8–18 of Example 20-7 could also have been written this way:

Example 20-9. K-Dimension Loop as Innermost Loop–A, a Highly Inefficient Approach

Example 1-8

```
for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
  tile<TILE_K/KPACK, TILE_N*KPACK> tB;
  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    tile<TILE_M, TILE_K*sizeof(type_t)> tA;
    for (int k = 0; k < K; k += TILE_K) {
      TILELOAD(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
      TILELOAD(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
      tdp(tC[m_acc][n_acc], tA, tB);
    }
  }
}
```

While both approaches yield correct results, there are K/TILE_K×N_ACC B tile loads are in the reference implementation. Additionally, K/TILE_K×N_ACC×M_ACC B tile loads in the implementation presented in this section. The number of A tile loads is identical.

This approach is also characterized by excessive pressure on the memory and an increased number of tile loads.

Suppose the B_mem data resides in main memory.

- In the reference implementation, a new chunk of TILE_K×TILE_N B data is read every M_ACC iteration of the inner loop.
- The inner loop then reuses the read data.

In the current implementation:

- When **n_acc == m_acc == 0**, a new chunk of TILE_K×TILE_N B data is read every iteration of the inner loop.
- Then the same data is read (presumably from caches) on subsequent iterations of n_acc, m_acc.
- This burst access pattern of reads from main memory results in increased data latency and decreased performance.

Hence, keeping the K-dimension loop outside the M_ACC and N_ACC loops is recommended.

Pre-Loading Innermost Loop Tiles

Consider the following replacement code for the code in lines 8–18 of Example 20-10:

Example 20-10. Innermost Loop Tile Pre-Loading

Example 1-9

```
1     for (int k = 0; k < K; k += TILE_K) {
2       tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
3       for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
4         TILELOAD(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
5       for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
6        tile<TILE_K/KPACK, TILE_N*KPACK> tB;
7         TILELOAD(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
8         for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
9           tdp(tC[m_acc][n_acc], tA[m_acc], tB);
10        }
11      }
12    }
```

- The A-tile has been extended to an array of A-tiles (line 2) and pre-read the A tiles for the current K loop iteration (lines 3–4).
- A pre-read A-tile is used in the tile multiplication (line 9).
- There were K/TILE_K×N_ACC×M_ACC A-tile reads in the reference implementation, while there are only K/TILE_K×M_ACC A-tile reads in the current implementation.

Hence, preallocation and pre-reading the tiles of the innermost loop (tA[M_ACC] in this case) is recommended.

The maximum number of tiles used at any given time in this scenario is N_ACC×M_ACC+M_ACC+1 instead of N_ACC×M_ACC+2 in the reference implementation. Since this optimization requires the preallocation of additional M_ACC-1 tiles and tiles are a scarce resource, if N_ACC<M_ACC, switching the order of the N_ACC and M_ACC loops might be beneficial.

This way, it is possible to allocate N_ACC-1<M_ACC-1 additional tiles:

Example 20-11. Switched Order of M_ACC and N_ACC Loops

```
for (int k = 0; k < K; k += TILE_K) {
  tile<TILE_K/KPACK, TILE_N*KPACK> tB[N_ACC];
  for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
    TILELOAD(tB[n_acc], B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    tile<TILE_M, TILE_K*sizeof(type_t)> tA;
    TILELOAD(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
      tdp(tC[m_acc][n_acc], tA, tB[n_acc]);
    }
  }
}
```

2D Accumulator Array vs. 1D Accumulator Array

Consider Example 20-10 with the following scenarios:

- N_ACC=2,M_ACC=2

- N_ACC=4,M_ACC=1

As stated before, the number of A tile loads in lines 3–11 is M_ACC, and the number of B tile loads is N_ACC. Thus, the total number of tile loads (M_ACC+N_ACC) is 4 in the first scenario vs. 5 in the second one (an increase of 25%), even though both scenarios perform the same amount of work.

Hence, using 2D accumulator arrays is recommended. Selecting dimensions close to square is strongly recommended (since x=y minimizes f(x,y)=x+y under the constraint x×y=const).

20.5.5.2 Software Pipelining of Tile Loads and Stores

It is a best practice to interleave instructions using different resources so they may be executed in parallel, preventing a bottleneck involving a specific resource. Therefore, preventing sequential TILELOADs and TILESTOREs (see lines 19–23 of Example 20-8 and lines 3–4 of Example 20-10) is recommended. Instead, interleave them with the TDP instructions (see Example 20-12).

20.5.5.3 Optimized GEMM Implementation

Below is the original code from Example 20-8, augmented with the insights from Example 20-10, with tile loads and stores interleaved with tdps:

Example 20-12. Optimized GEMM Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2   for (int m = 0; m < M; m += M_ACC*TILE_M) {
3     tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4     tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5     tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7     for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8       for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9         TILEZERO(tC[m_acc][n_acc]);
10
11    for (int k = 0; k < K; k += TILE_K) {
12      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
13        TILELOAD(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
14        for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
15          if (n_acc == 0)
16            TILELOAD(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
17          tdp(tC[m_acc][n_acc], tA[m_acc], tB);
18          if (k == K - TILE_K) {
19            int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
20            TILESTORE(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
21          }
22        }
23      }
24    }
25  }
26 }
```

While placing the tile loads and stores under conditions inside the main loop (lines 13, 16, 20), conditions can be eliminated by sufficiently unrolling the loops.

The rest of this section presents a specific example of GEMM implemented in low-level Intel AMX instructions. This is to show a full performance potential from using Intel AMX extensions.

Example 20-13. Dimension of Matrices, Data Types, and Tile Sizes

```
#define M 32
#define K 128
#define N 32
#define M_ACC 2
#define N_ACC 2
#define TILE_M 16
#define TILE_K 64
#define TILE_N 64

typedef int8_t type_t
typedef int32_t res_type_t
```

The following code is an example of the algorithm outlined in Example 20-13.

Example 20-14. Optimized GEMM Assembly Language Implementation

```
1    typedef struct {
2          uint8_t palette_id;
3          uint8_t startRow;
4          uint8_t reserved[14];
5          uint16_t cols[16];
6          uint8_t rows[16];
7    } __attribute__ ((__packed__)) TILECONFIG_t;
8
9    static const TILECONFIG_t tc = {
10         1,                                          // palette_id
11         0,                                          // startRow
```

Lines 1-12 in Example 20-14 define the tile configuration for this example and contain information about tile sizes. Tile configuration should be loaded before executing Intel AMX instructions (line 16). Tile sizes are defined by the configuration at load time and can't be changed dynamically (unless TILERELEASE is called). The 'palette_id' field in the configuration specifies the number of logical tiles available; palette_id == 1 means 8 logical tiles are available, named tmm0 through tmm7 are available. This example uses seven logical tiles (tmm4, tmm5 for A, tmm6 for B, tmm0-tmm3 for C).

According to the dimensions specified, the K-loop consists of two iterations (cf. code listing 8.1, line 11) according to the dimensions specified in the example. Lines 23-34 implement the first iteration, and lines 35-46 the second iteration. Note the interleaving of the tdp and TILESTORE instructions to hide the high cost of TILESTORE operation.

Variable Input Dimensions

The code in Example 20-12 and 20-14 process an entire matrix of inputs of size MxK. Sometimes, only part of the input is significant, so it is beneficial to adapt the computation to the actual input size. Often, topologies using self-attention it is enough to process only the first significant m rows of the input, where m < M. For example, taking the GEMM dimensions described above with the choice of a 1D accumulator array of N_ACC=2,M_ACC=1, when accepting data as input with at most sixteen significant rows, we can degenerate the m loop (line 2 in Example 20-12) essentially reducing the computation by half.

Notably, in variable M dimension use cases there is an advantage to 1D accumulators. Up to N_ACC=6, M_ACC=1 dimensions are possible if N is 96 or larger, one tile for A, one for B, and six for the accumulator.

20.5.5.4 Direct Convolution with Intel® AMX

Direct convolution is performed directly on the input data; no data replication is required. However, there are some layout considerations.

Activations Layout

Similar to the Intel DL Boost use case, the activations are laid out in a layout obtained from the original layout by the following procedure:

Example 20-15. Activations Layout Procedure

```
#define K C                          // K-dimension of the A matrix = channels
#define M H*W                        // M-dimension of the A matrix = spatial
type_t A_mem_orig[C][H][W];          // Original activations tensor
type_t A_mem[H][W][K];               // Re-laid A matrix7

for (int c = 0; c < C; ++c)
  for (int h = 0; h < H; ++h)
    for (int w = 0; w < W; ++w)
      A_mem[h][w][c] = A_mem_orig[c][h][w];
```

This procedure on the left side of the diagram below shows the conversion of a 3-dimensional tensor into a 2-dimensional matrix:

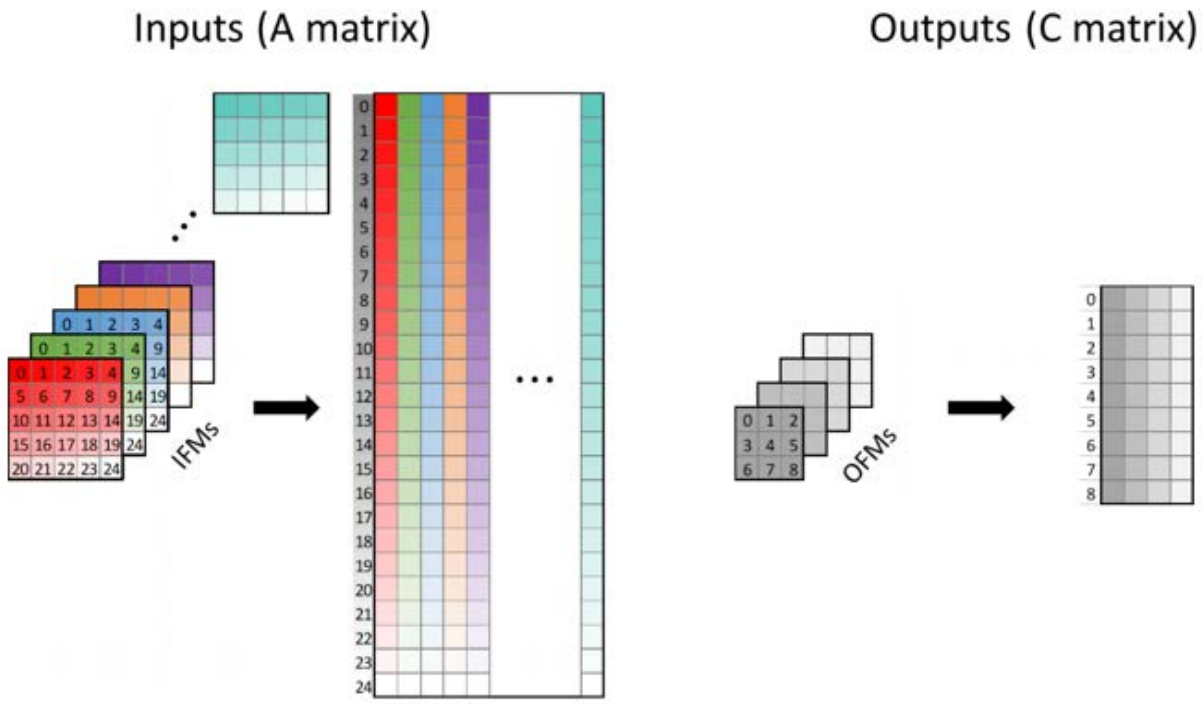


Figure 20-3. Activations layout

The procedure shown on the right is identical for the outputs (for example, the activations of the next layer in the topology).

Weights Layout

Similar to the Intel DL Boost use case, the weights are re-laid by the following procedure:

Example 20-16. Weights Re-Layout Procedure

```
#define KH ...                                  // Vertical dimension of the weights
#define KW ...                                  // Horizontal dimension of the weights
#define KPACK (4/sizeof(type_t))                // Vertical K packing into Dword

type_t B_mem_orig[K][N][KH][KW];                // Original weights
type_t B_mem[KH][KW][K/KPACK][N][KPACK];        // Re-laid B matrices

for (int kh = 0; kh < KH; ++kh)
  for (int kw = 0; kw < KW; ++kw)
    for (int k = 0; k < K; ++k)
      for (int n = 0; n < N; ++n)
        B_mem[kh][kw][k/KPACK][n][k%KPACK] = B_mem_orig[k][n][kh][kw];
```

The procedure transforms the original 4-dimensional tensor into a series of 2-dimensional matrices (a single matrix is highlighted in orange in Example 20-16) as illustrated in Figure 20-8 for **KH=KW=3**, resulting in a series of nine B-matrices:

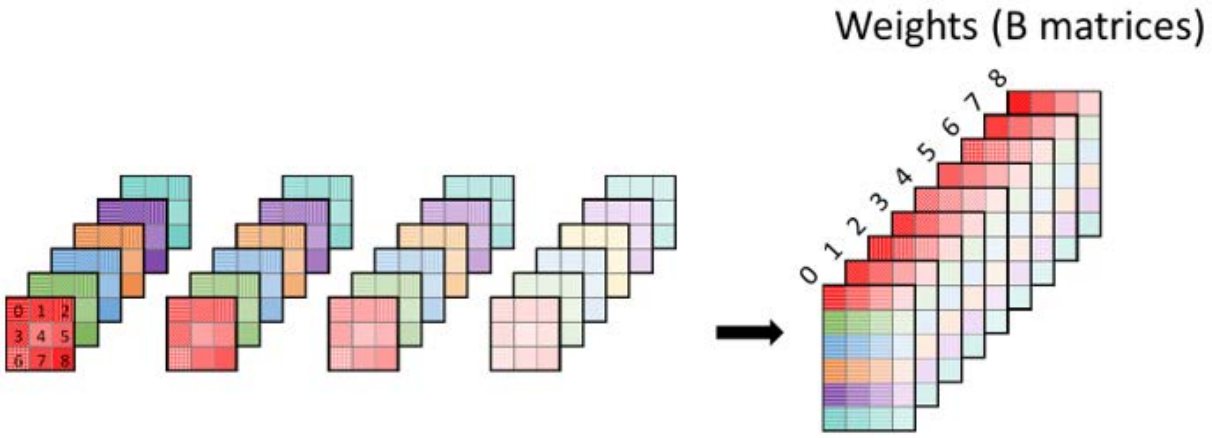


Figure 20-4. Weights Re-Layout

20.5.5.5 Convolution - Matrix-like Multiplications and Summations Equivalence

Figure 20-5 illustrates the equivalence between convolution and summation of a series of matrix-like multiplications between subsets of the 2-dimensional A-matrix representing the 3-dimensional activations tensor. The 2-dimensional B-matrices correspond to the various spatial elements of the weights filter.

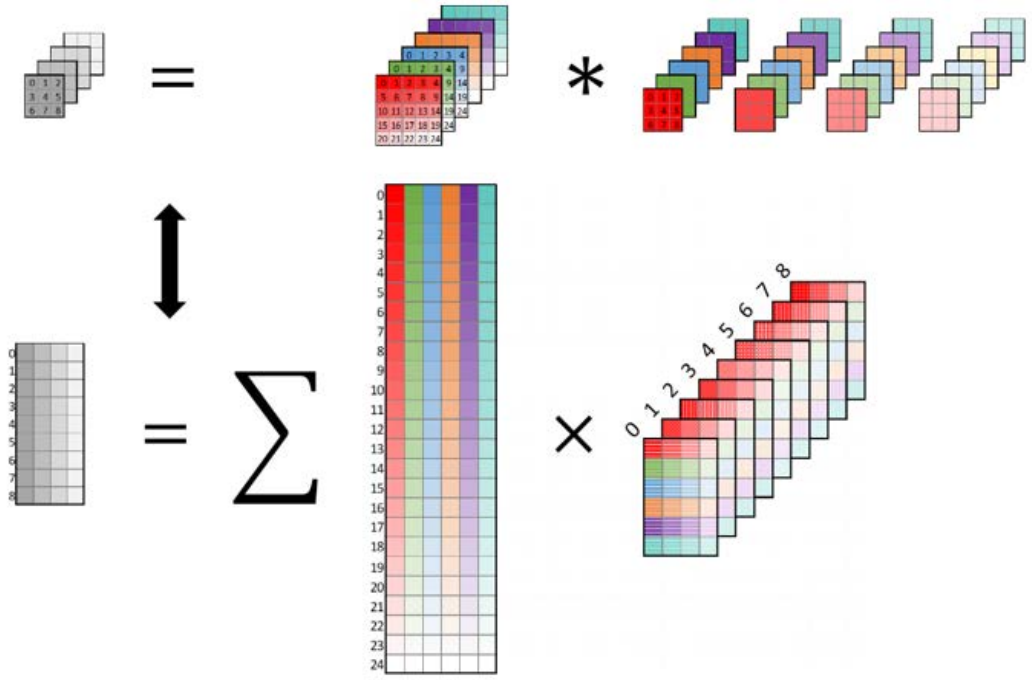


Figure 20-5. Convolution-Matrix Multiplication and Summation Equivalence

The A-matrix subset participating in the matrix-like multiplication depends on the spatial weight element in question (i.e., the kh,kw coordinates, or the index in the range 0–8 in the previous example). For each weight element, the A-matrix's participating rows will interact with the weight element when the filter is slid over the activations. For

example, when sliding the filter over the activations in the previous example, weight element 0 will only interact with activation elements 0, 1, 2, 5, 6, 7, 10, 11, and 12. For example, it will not interact with activation element four because when the filter is applied in such a manner (i.e., weight element 0 interacts with activation element 4), weight elements 2, 5, and 8 leave the activation frame entirely. The A-matrix subsets for several weight elements are illustrated in the following figure.

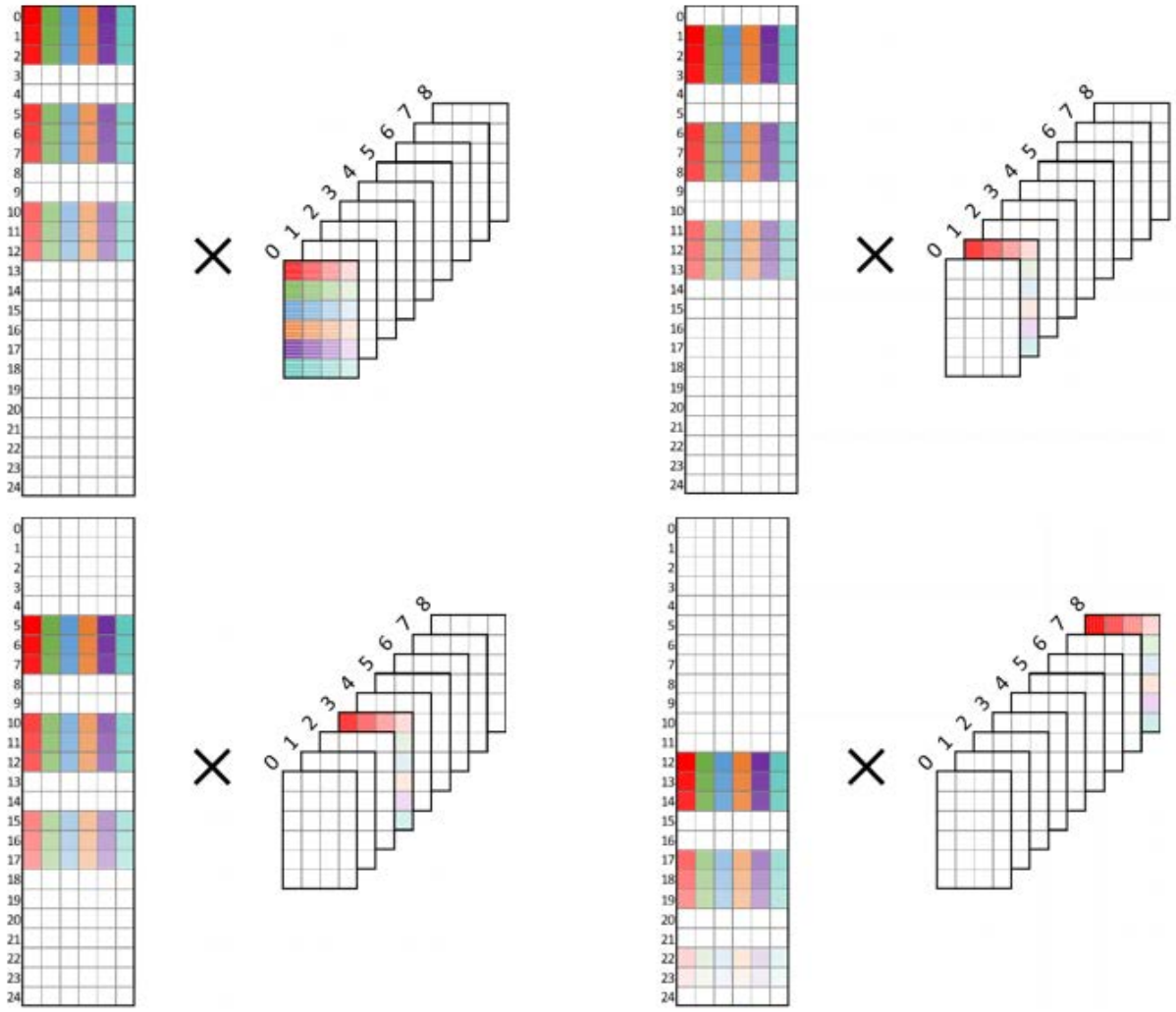


Figure 20-6. Matrix-Like Multiplications Part of a Convolution

20.5.5.6 Optimized Convolution Implementation

Replace the common defines in Example 20-17 with the following:

Example 20-17. Common Defines for Convolution

```
#define H ...                   // The height of the activation frame
#define W ...                   // The width of the activation frame
#define MA (H*W)                // The M dimension (rows) of the A matrix
#define K ...                   // Number of activation channels
#define N ...                   // Number of output channels
#define KH ...                  // The height of the weights kernel
#define KW ...                  // The width of the weights kernel
#define SH ...                  // The vertical stride of the convolution
#define SW ...                  // The horizontal stride of the convolution
#define M_ACC ...               // Number of C accumulators spanning the M dimension
#define N_ACC ...               // Number of C accumulators spanning the N dimension
#define TILE_M ...              // Number of rows in an A or C tile
#define TILE_K ...              // Number of columns in an A tile or rows in a B tile
#define TILE_N ...              // Number of columns in a B or C tile

#define HC ((H-KH)/SH+1)        // The height of the output frame
#define WC ((W-KW)/SW+1)        // The width of the output frame
#define MC (HC*WC)              // The M dimension (rows) of the C matrix

typedef ... type_t;             // The type of the data being operated on
typedef... res_type_t;          // The data type of the result

#define KPACK (4/sizeof(type_t))                    // Vertical K packing into Dword

type_t A_mem[H][W][K];                              // A matrix (equivalent to A_mem[H*W][K])
type_t B_mem[KH][KW][K/KPACK][N][KPACK];            // B matrices
res_type_t C_mem[MC][N];                            // C matrix

template<size_t rows, size_t cols> class tile;
template<class T> void TILEZERO (T& t);
template<class T> void TILELOAD (T& t, void* src, size_t stride);
template<class T> void TILESTORE(T& t, void* dst, size_t stride);
template<class TC, class TA, class TB> void tdp(TC& tC, TA& tA, TB& tB);

int mc_to_ha(int mc) {return mc / HC * SH;}     // C matrix M -> A tensor h coord
int mc_to_wa(int mc) {return mc % HC * SW;}     // C matrix M -> A tensor w coord
```

Replace the implementation in Example 20-17 with the following:

Example 20-18. Optimized Direct Convolution Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2  for (int m = 0; m < MC; m += M_ACC*TILE_M) {
3   tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4   tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5   tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7   for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8    for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9     TILEZERO(tC[m_acc][n_acc]);
10
11  for (int k = 0; k < K; k += TILE_K) {
12   for (int kh = 0; kh < KH; ++kh) {
13    for (int kw = 0; kw < KW; ++kw) {
14     for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
15      int nc = n + n_acc*TILE_N;
16      TILELOAD(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
17      for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
18       int mc = m + m_acc*TILE_M;
19       if (n_acc == 0) {
20        int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
21        TILELOAD(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
22       }
23       tdp(tC[m_acc][n_acc], tA[m_acc], tB);
24       if (k + kh + kw == K - TILE_K + KH + KW - 2)
25        TILESTORE(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
26      }
27     }
28    }
29   }
30  }
31 }
32 }
```

The divergences in blue include:

- The loop over the M-dimension (line 2) references the M-dimension of the C-matrix (since the M-dimensions of A and C no longer have to be the same). To get the corresponding A-matrix m index from a C-matrix m index, one must employ the conversion functions mc_to_ha() and mc_to_wa() (line 20).
- There are additional loops over the weights kernel dimensions KH and KW (lines 12–13), which define the B-matrix to be used (line 16), enter into the condition for accumulator tile storing (line 24) and computation of A-matrix coordinates (line 20).
- The stride of the A tile load must account for the convolutional horizontal stride (line 21).

Note that care should be taken to define TILE_M*M_ACC so that it cleanly divides WC (the width of the output frame), i.e., WC%(TILE_M*M_ACC)==0. Otherwise, some tiles will end up loading data that should not be multiplied by the corresponding weight element (see Figure 20-6). Possible mitigations of this issue:

- An M_ACC loop with a dynamic upper limit depending on the current position in A.

- Use different sized A tiles (and correspondingly C tiles) depending on the current position in A (if there are enough free tiles, performing TILECONFIG during the convolution is highly discouraged).
- Define TILE_M without consideration for WC and remove/disregard the "junk" data from the results at the post-processing stage (code not shown). Care should be taken in this case concerning the advancement of the m index (line 2) since the current assumption is that every row of every tile is valid (corresponds to a row in the C matrix). This is no longer true if "junk" data is loaded: a C-tile will have less than TILE_M rows of C.

Location of the KH,KW Loops

As shown in Example 20-9, putting the loop over the K-dimension inside an inner M_ACC or N_ACC loop would be injudicious.

The same considerations hold in the case of the kh,kw loops. While there is no functional obstacle precluding the positioning of the kh,kw loops further up (before lines 12-13), it is recommended to keep them under the K loop and above the M_ACC, N_ACC loops. This is because, during the traversal of kh,kw with the same k value, the TILELOAD of A-data (line 21) will have much overlap with A-data loaded for previous values of kh,kw (with the same k value). This data will likely reside in the lowest-level cache. Moving the kh,kw loops upward will reduce that likelihood.

20.6 CACHE BLOCKING

Data movement costs vary greatly depending on where the data lies in the cache hierarchy. When the matrices involved in a GEMM or convolution are larger than the available cache, computations must proceed in such a manner as to optimize data reuse from the cache. A simple cache-blocking scheme is implemented to simultaneously process partial blocks of the A, B, and C matrices.

20.6.1 OPTIMIZED CONVOLUTION IMPLEMENTATION WITH CACHE BLOCKING

The following example focuses on implementing cache blocking for the optimized convolution implementation. However, note that similar changes can also be made to the optimized GEMM implementation. Alternatively, the GEMM implementation can be derived as a special case of convolution with KH=KW=1 and SH=SW=1.

In addition to the common defines in Example 20-18, add the following:

Example 20-19. Additional Defines for Convolution with Cache Blocking

```
#define MC_CACHE ...                                    // Extent of cache block along the M dimension of the C matrix
#define K_CACHE ...                                     // Extent of cache block along the K dimension
#define N_CACHE ...                                     // Extent of cache block along the N dimension
typedef ... acc_type_t;                                 // The accumulation data type (either int32 or float)
acc_type_t aC_mem[M_ACC][N_ACC][TILE_M][TILE_N];                // Accumulator buffers of C
```

Replace the implementation in Example 20-19 with the following:

Example 20-20. Optimized Convolution Implementation with Cache Blocking

```
1 for (int nb = 0; nb < N; nb += N_CACHE) {
2  for (int mb = 0; mb < MC; mb += MC_CACHE) {
3   for (int kb = 0; kb < K; kb += K_CACHE) {
4    for (int n = nb; n < nb + N_CACHE; n += N_ACC*TILE_N) {
5     for (int m = mb; m < mb + MC_CACHE; m += M_ACC*TILE_M) {
6      tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
7      tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
8      tile<TILE_K/KPACK, TILE_N*KPACK> tB;
9
10      for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
11       for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
12        if (kb == 0)
13         TILEZERO(tC[m_acc][n_acc]);
14        else {
15         int m_aC = (m - mb) / TILE_M + m_acc;
16         int n_aC = (n - nb) / TILE_N + n_acc;
17         TILELOAD(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
18              TILE_N*sizeof(acc_type_t));
19        }
20
21      for (int k = kb; k < kb + K_CACHE; k += TILE_K) {
22       for (int kh = 0; kh < KH; ++kh) {
23        for (int kw = 0; kw < KW; ++kw) {
24         for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
25          int nc = n + n_acc*TILE_N;
26          TILELOAD(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
27          for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
28           int mc = m + m_acc*TILE_M;
29           if (n_acc == 0) {
30            int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
31            TILELOAD(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
32           }
33           tdp(tC[m_acc][n_acc], tA[m_acc], tB);
34           if (k + kh + kw == K - TILE_K + KH + KW - 2)
35            TILESTORE(tC[m_acc][n_acc], &C_mem[mc][nc],
36                 N*sizeof(res_type_t));
37           else if (k + kh + kw == kb + K_CACHE - TILE_K + KH + KW - 2) {
38            int m_aC = (m - mb) / TILE_M + m_acc;
39            int n_aC = (n - nb) / TILE_N + n_acc;
40            TILESTORE(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
41                 TILE_N*sizeof(acc_type_t));
42           }
43          }
44         }
45        }
46       }
```

Example 20-20. (Contd.)Optimized Convolution Implementation with Cache Blocking

```
47        }
48      }
49    }
50  }
51 }
52 }
```

The loops over the N, MC, and K-dimensions are replaced by loops over cache blocks of N, MC, and K.

Additional loops over the entire N, MC, and K dimensions are added at the outermost level. These loops have a step size equal to N, MC, and K cache blocks.

In the case of cache-blocking along the K dimension, additional calls to TILELOAD and TILESTORE are required to load and store intermediate accumulation results. Note that this adds additional memory traffic, especially for int8 output data types (as Accumulation data type is either int32_t or float). For this reason, it is generally not advisable to block along the K dimension.

For simplicity, assume the following relationships:

- N is an integer multiple of N_CACHE: an integer multiple of N_ACC*TILE_N.
- MC is an integer multiple of MC_CACHE: an integer multiple of M_ACC*TILE_M. As before, the condition WC%(TILE_M*M_ACC)==0 still holds.
- K is an integer multiple of K_CACHE: an integer multiple of TILE_K.

Define the following set of operations as the compute kernel of the optimized convolution implementation. First, initialize the accumulation tiles to zero (line 13) for a M_ACC*TILE_M x N_ACC*TILE_N chunk of the C-matrix. Next, for each of the KH*KW B-matrices, the matrix multiplication of the corresponding M_ACC*TILE_M x K chunk of the A-matrix by a K x N_ACC*TILE_N chunk of the B-matrix is performed, each time accumulating to the same set of accumulation tiles (lines 18–30). Finally, the results are stored in the C-matrix (line 32).

Continue with the computation **of a full cache block** of C-matrix, ignoring any blocking along the K dimension. First, the kernel is performed for the first chunks of the A, B, and C cache blocks. Next, the chunks of A and C advance along the M dimension, and the kernel is repeated with the same chunk set of the B-matrices. The above step is repeated until the last chunks of A and C in the current cache block have been accessed. Next, the chunks of B and C are advanced along the N-dimension by N_ACC*TILE_N, and the chunk of A returns to the beginning of its cache block.

Observe the following from the above description of the computation of a **full cach**e block of the C-matrix:

- For each kernel iteration, it is better if the current chunk of matrix A (roughly KH*M_ACC*TILE_M*K*sizeof(type_t)) fits into the DCU. This allows for maximal data reuse between the partially overlapping regions of A that need to be accessed by the different B matrices.
- Advancing from one chunk of matrix A to the next, it is better if the current chunk set of the B matrices (in total, KH*KW*K*N_ACC*TILE_N*sizeof(type_t)) fits into the DCU.
- Advancing from one chunk set of the B matrices to the next, it is better if the current cache block of matrix A fits into the MLC.
- Advancing from one cache block of matrix A to the next, it is better if the current cache block of the B matrices (in total, KH*KW*K*N_CACHE*sizeof(type_t)) fits into the MLC.

From these observations, a general cache-blocking strategy is choosing MC_CACHE and N_CACHE to be as large as possible while keeping the A, B, and C cache blocks in the MLC.

Intel® AMX-Specific Considerations

A specific feature of Intel AMX-accelerated kernels to keep in mind when applying the previous cache-blocking recommendations is any post-processing of results from the Intel AMX unit (e.g., adding bias, dequantizing,

converting between data types) must occur by way of vector registers. Thus, a buffer is needed to store results from the accumulation tiles and load them into vector registers for post-processing. Note that if **acc_type_t** is the same as **res_type_t**, the C-matrix can store intermediate results. However, the buffer is small (at most 4KB for the accumulation strategies described in "2D Accumulator Array vs. 1D Accumulator Array") and easily fits into the DCU. While it should still be considered when determining the optimal cache block partitioning, it is unlikely to influence kernel performance strongly.

20.7 MINI-BATCHING IN LARGE BATCH INFERENCE

Layers have different sizes and shapes, which require different cache and memory-blocking strategies. There are layers with a small spatial dimension (M) and relatively larger shared dimension (K) and SIMD dimension (N). In such layers, the weights are significantly larger than the inputs. Therefore, most load operations are weights matrix loads whose cost is high when the weights reside in memory or last level cache.

Running a large batch allows employing an optimization that amortizes the cost of loading the weight matrix. The idea is to use the same weights for multiple inputs, e.g., execute the same layer with multiple images. This optimization is highly applicable in CNNs where the inputs of the first layers are large while the weights are relatively small but end with small input images and large weight matrices. Optimal execution of the topology starts in the instance or image affinity, where a single input goes through one layer after another before the next input is retrieved. At some point, the topology execution switches to layer affinity, where the same layer processes several inputs (mini-batch) before continuing to the next layer.

For example, in ResNet-50, the conv-1 to conv-4 layers have relatively large IFMs and smaller weight matrices. However, many weight matrices are larger than MLC size (mid-level cache) in the conv-5 layers. The switchover point from image affinity to layer affinity on a 4th Generation Intel® Xeon® Processor microarchitecture is the first layer of conv-5.

The diagram below illustrates six layers with four instances per thread (mini-batch of four). Boxes with identical colors identify the same layers in each column. Arrows flowing downward through each column's layers represent the data flow of a particular instance. Translucent red arrows identify the execution order of layers with corresponding instances. The first four layers of the diagram have instance (aka image) affinity, and the last two have layer affinity.

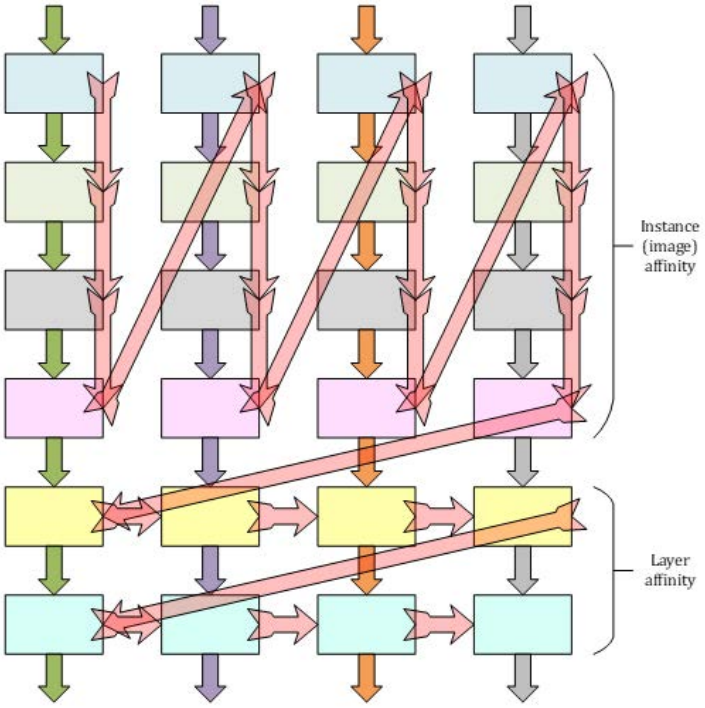


Figure 20-7. Batching Execution Using Six Layers with Four Instances Per Thread

On Resnet-50, this optimization can yield a 17% performance gain.

20.8 NON-TEMPORAL TILE LOADS

When a regular tile load is issued, the data for the tile are placed in L2, L1, and then in the tile register (DRAM/L3->L2->L1->tile register), as with any other register load. This has the well-known benefit of reduced data read latency due to data proximity when recently accessed data are reaccessed after a short time. However, indiscriminate application of this approach might sometimes prove detrimental.

Consider the code in Example 20-4, referring to the unoptimized, unblocked implementation for simplicity. The five loops in the code listing alongside the total input (A) matrix data and weights (B) matrix data accessed at each loop level are shown in Table 20-3. The original row in the code listing is provided for convenience:

Table 20-3. Five Loops in Example 20-20

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	M×K	K×N
2	m	[0:M:M_ACC×TILE_M]	M×K	K×N_ACC×TILE_N
8	k	[0:K:TILE_K]	MC_CACHE×K	
9	n acc	[0:N_ACC:1]	M_ACC×TILE_M×TILE_K	TILE_K×N_ACC×TILE_N
12	m ac	[0:M_ACC:1]		

20.8.1 PRIORITY INVERSION SCENARIOS WITH TEMPORAL LOADS

For the following discussion, assume:

- The data type is int8 (i.e., each element in the table above takes 1 byte).
- TILE_M=16, TILE_K=64, TILE_N=16 (i.e., all tiles are of size 1kB).
- L1 cache size is 32kB.
- M_AC=N_ACC=2.

Scenario One:

Consider the following scenario, including M=256, K=1024, and N=256.

Table 20-4 illustrates accessed data sizes:

Table 20-4. Accessed Data Sizes: Scenario One

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	256kB	256kB
2	m	[0:M:M_ACC×TILE_M]		32kB
8	k	[0:K:TILE_K]	32kB	
9	n acc	[0:N_ACC:1]	32kB	2kB
12	m ac	[0:M_ACC:1]		

At the k loop level, the combined sizes of A and B accessed data will overflow the L1 cache by a factor of two. Proceeding to the m-level, since m is progressing, new A-data are constantly read (a total of 256kB-32kB=224kB new A data), while the same 32kB of B data are being accessed repeatedly. Thus, a priority inversion occurs: new A-data placed in the L1 cache repeatedly are accessed only once. They evict the 32kB of B data that are accessed eight times. Placement of A data in the L1 cache is not beneficial: the next time the same data are accessed will be in the n loop after 256kB (x8 L1 cache size) of A data has been read. Additionally, it is detrimental because it causes repeated eviction of 32kB of B data that could have been read from the L1 cache eight times.

Scenario Two:

Consider the following scenario, including M=32, K=1024, and N=256. Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Table 20-5. Accessed Data Sizes: Scenario Two

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	32kB	256kB
2	m	[0:M:M_ACC×TILE_M]		32kB
8	k	[0:K:TILE_K]		

Table 20-5. Accessed Data Sizes: Scenario Two (Contd.)

Row	Var	Variable Range	A Data Size	B Data Size
9	n acc	[0:N_ACC:1]	2kB	2kB
12	m ac	[0:M_ACC:1]		

These two basic scenarios can be readily extended to the blocked code in Example 20-20.

Table 20-6. Accessed Data Sizes Extended to Blocked Code

Row	Var	Variable Range	A Data Size	B Data Size
1	nb	[0:N:N_CACHE]	M×K	
2	mb	[0:MC:MC_CACHE]	M×K	
3	kb	[0:K:K_CACHE]	MC_CACHE×K	
4	n	[nb:nb+N_CACHE:N_ACC×TILE_N]	MC_CACHE×K_CACHE	K_CACHE×KH×KW×N_ACC×TILE_N
5	m	[mb:mb+MC_CACHE:M_ACC×TILE_M]		
18	k	[kb:kb+K_CACHE:TILE_K]		
19	kh	[0:KH:1]	/*/*	TILE_K×KH×KW×N_ACC×TILE_N
20	kw	[0:KW:1]	M_ACC×TILE_M×TILE_K	
21	n acc	[0:N_ACC:1]		TILE_K×N_ACC×TILE_N
24	m ac	[0:M_ACC:1]		

NOTE

Due to the nature of convolution, the loops over kh, kw reuse most of the A-data.

The innermost loops m_acc, n_acc, kh,kw will access at most M_ACC kB of A data and KH×KW×N_ACC kB of B-data, which, in some cases (e.g., KH=KW=3, N_ACC=4) might already overflow the L1 cache size. Thus, several opportunities for priority inversions exist in this more complex loop structure, depending on the parameters in the table above:

- B-data evicting reusable A-data at the kh,kw loops level.
- A-data evicting reusable B-data at the m loop level.
- B-data evicting reusable A-data at the n loop level.
- A-data evicting reusable B-data at the mb loop level.
- B-data evicting reusable A-data at the nb loop level.

Solution to Priority Inversions: Non-Temporal Loads

Intel AMX architecture introduces a way to load tile registers bypassing the L1 cache via non-temporal tile loads (TILELOADDT1). This allows the user to deal with priority inversions such as those described above by loading the large, non-reusable data chunk with non-temporal loads. Thus, the larger chunk is prevented from evicting the smaller, frequently used data chunk. In Table 20-4, the A-tiles are loaded with non-temporal loads while loading B-tiles with temporal loads. This ensures that the B-tile loads at the m loop level will come from the L1 cache. In Table 20-5, the B-tiles are loaded with non-temporal loads while loading A-tiles with temporal loads, thus ensuring that the A-tile loads at the n loop level will come from the SL1 cache.

20.9 USING LARGE TILES IN SMALL CONVOLUTIONS TO MAXIMIZE DATA REUSE

A convolution with a small-sized input frame can make the Intel AMX computation inefficient.

Consider the following example: a 7x7 input frame, with padding of 1 (size including padding is 9x9), convolved with a 3x3 filter to produce a 7x7 output frame.

Figure 20-8 shows the pieces participating in the convolution (in yellow) interacting with the khaki=0,0 weight element.

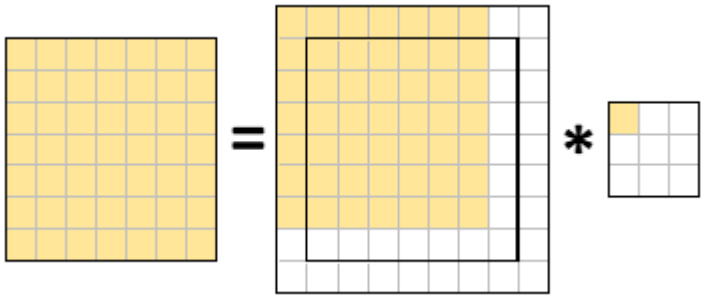


Figure 20-8. A Convolution Example

Thus, the yellow parts of the input frame are the only ones that should be loaded into A-tiles when processing weight element kh,kw=0,0. The white parts of the input frame should be ignored. This requires the number of tile rows to be set at seven, utilizing less than half of the A-tile, reducing B (weights) data reuse by a factor of two. Each A-tile is now half the size, and seven tiles are required to cover the spatial dimension. Because there are not seven tiles, B-tiles must be loaded twice as many times, potentially leading to significant performance degradation, depending on the size of the weights. This is usually inversely proportional to the spatial size of the input frame).

Figure 20-9 shows three A-tiles with sixteen rows and one tile with seven rows to cover the entire spatial dimension of the convolution.

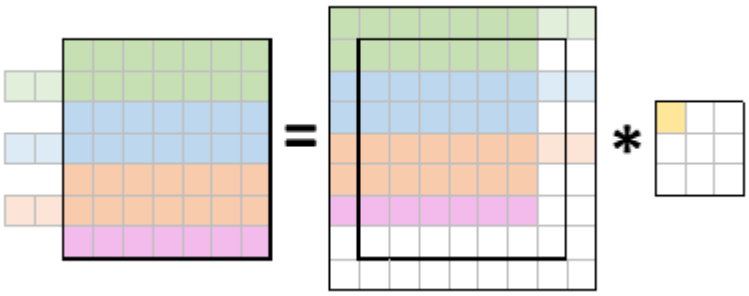


Figure 20-9. A Convolution Example with Large Tiles.

Each tile is highlighted differently. The green, blue, and orange tiles now load those two “extra” pieces previously ignored. Those pieces will waste compute resources and take up two rows in the accumulator tiles. The user may

ignore those rows in subsequent computations (e.g., int8-quantization, RELU, etc.), complicating the implementation. The potential benefit of increased B-data reuse could be dramatic, however.

20.10 HANDLING INCONVENIENTLY-SIZED ACTIVATIONS

Occasionally, the spatial dimensions of an activation might be ill-suited for efficient tiling with tiles. Consider a GEMM with activations' M=100. This poses a challenge: while the M dimension can be neatly tiled by ten tiles, each with ten rows, this approach is inefficient since a larger M dimension of 112 requires only seven tiles with sixteen rows. This means that the data reuse for M=100 is 30% worse than for M=112.

The following solutions will be useful:

1. Define two types of A- and C-tiles – tiles with 16 rows and one tile with four. Use tiles of the first type for M=0..9 and the second type tile for M=96..99.
2. Allocate extra space in A and C buffers, as if M=112, and use tiles with 16 rows exclusively. The extra space need not be zeroed out or otherwise prepared in any way. In this case, the last (seventh) tile will load four meaningful rows (M=96..99) and twelve "garbage" rows (M=100..111). At the output, tile C will have four meaningful rows (M=96..99) and twelve "garbage" rows (M=100..111) which the user can then ignore.

The first solution does not require tampering with the A and C buffers and computes 100 tile rows, producing a clean result. Still, it requires additional A- and C-tiles unused throughout the computation except at the end. Since only eight tiles are available, this requirement can be costly and might **reduce** the data reuse (e.g., to use a 2D accumulator array, you would need three x2 C-tiles, two A-tiles, and two B-tiles, equaling ten tiles). The second solution avoids this requirement by complicating buffer handling and paying with additional loads, compute, and storing (it loads, computes, and stores 112 tile rows).

20.11 POST-CONVOLUTION OPTIMIZATIONS

Most Intel AMX-friendly applications are from the Deep Learning domain, where the data flows through multiple layers. It is often necessary to process the convolution output before passing it as an input to the next layer (processing operations depend on a specific application).

This stage is called **post-convolution**.

20.11.1 POST-CONVOLUTION FUSION

As with Intel AVX-512 code, a critical optimization is the "fusion" of post-convolutional operations to the convolutional data they operate upon. Fusion reduces the memory hierarchy thrashing. Additionally, fusing the quantization step gains x2 (for bfloat16 data type) or x4 (for int8 data type) compute bandwidth and reduces memory bandwidth by x2 or x4, respectively.

Consider the code in Example 20-12. Lines 7-24 contain the entire GEMM operation for any M, N coordinates in the output. Thus, the optimal location to post-process the data computed in lines 7-24 is right before line 24 while it is still in the low-level cache.

In Example 20-21, the blue code illustrates a fully unrolled example from lines 7 through 24, for int8 GEMM with K=192, N_ACC=M_ACC=2, TILE_M=2, TILE_K=64, TILE_N=16. The convolution code is fused with the post-convolution code (blue) that quantizes the output and ReLU. To keep the post-convolution code in the example short, an unrealistically low value of TILE_M=2 was chosen.

In that example, an additional buffer, temporary_C, contains the convolutional results of M_ACCxN_ACC tiles. The results are stored at the end of the convolutional part and loaded during the post-convolutional part. A temporary

buffer is required because the size of the post-processed data is four times smaller. Hence, the convolutional output cannot be written directly to the output buffer.

The GPRs r8, r9, r10, r11, and r14 point to the current location in the A, B, C, temporary_C, and q_bias (which holds the quantization factors and biases) buffers, respectively.

The macros A_OFFSET(m,k), B_OFFSET(k,n), C_OFFSET(m,n), C_TMP_OFFSET(m,n), Q_OFFSET(n), and BIAS_OFFSET(n) receive as arguments m,k,n tile indices and return the offset of the data from r8,r9,r10, r11, and r14, respectively.

Example 20-21. Convolution Code Fused with Post-Convolution Code

```
1   #define TILE_N_B            (N)
2   #define A_OFFSET(m,k)       ((m)*K*TILE_M + (k)*TILE_K)
3   #define B_OFFSET(k,n)       ((k)*N*TILE_N*4 + (n)*TILE_N*4)
4   #define C_OFFSET(m,n)       ((m)*N*TILE_M + (n)*TILE_N)
5   #define C_TMP_OFFSET(m,n)((m)*N*TILE_M*4 + (n)*TILE_N*4)
6   #define Q_OFFSET(n)         ((n)*TILE_N*4)
7   #define BIAS_OFFSET(n)      ((n)*TILE_N*4 + N*4)
8
9   static const TILECONFIG_t tc = {
10  1,                                                          // Palette ID
11  0,                                                          // Start row
12  0,  0,  0,  0,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0,           // Reserved – must be 0
13  64, 64, 64, 64, 64, 64, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0,     // Cols for 7 tiles used
14  2, 2, 2, 2, 2, 16, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0           // Rows for tiles used: 2 for A, C,
15                                                              // 16 for B
16   };
17
18  ldtilecfg    tc                                             // Load tile config
19  mov          r12, 192                                       // A stride
20  mov          r13, 128                                       // B, C_TMP stride
21  TILELOADD    tmm5, [r9 + r13*1 + B_OFFSET(0,0)]             // Load B [k,n] = [0,0]
22  TILELOADD    tmm4, [r8 + r12*1 + A_OFFSET(0,0)]             // Load A [m,k] = [0,0]
23  TILEZERO     tmm0                                           // Zero acc [m,n] = [0,0]
```

Example 20-21. Convolution Code Fused with Post-Convolution Code (Contd.)

```
24  tdpbusd         tmm0, tmm4, tmm5
25  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(0,1)]      // Load B [k,n] = [0,1]
26  TILEZERO        tmm2                                    // Zero acc [m,n] = [0,1]
27  tdpbusd         tmm2, tmm4, tmm6
28  TILELOADD       tmm4, [r8 + r12*1 + A_OFFSET(1,0)]      // Load A [m,k] = [1,0]
29  TILEZERO        tmm1                                    // Zero acc [m,n] = [1,0]
30  tdpbusd         tmm1, tmm4, tmm5
31  TILEZERO        tmm3                                        // Zero acc [m,n] = [1,1]
32  tdpbusd         tmm3, tmm4, tmm6
33  TILELOADD       tmm5, [r9 + r13*1 + B_OFFSET(1,0)]          // Load B [k,n] = [1,0]
34  TILELOADD       tmm4, [r8 + r12*1 + A_OFFSET(0,1)]          // Load A [m,k] = [0,1]
35  tdpbusd         tmm0, tmm4, tmm5
36  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(1,1)]          // Load B [k,n] = [1,1]
37  tdpbusd         tmm2, tmm4, tmm6
38  TILELOADD       tmm4, [r8 + r12*1 + A_OFFSET(1,1)]          // Load A [m,k] = [1,1]
39  tdpbusd         tmm1, tmm4, tmm5
40  tdpbusd         tmm3, tmm4, tmm6
41  TILELOADD       tmm5, [r9 + r13*1 + B_OFFSET(2,0)]          // Load B [k,n] = [2,0]
42  TILELOADD       tmm4, [r8 + r12*1 + A_OFFSET(0,2)]          // Load A [m,k] = [0,2]
43  tdpbusd         tmm0, tmm4, tmm5
44  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0     // Store C tmp [m,n] = [0,0]
45  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(2,1)]          // Load B [k,n] = [2,1]
46  tdpbusd         tmm2, tmm4, tmm6
47  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2     // Store C tmp [m,n] = [0,1]
48  TILELOADD       tmm4, [r8 + r12*1 + A_OFFSET(1,2)]          // Load A [m,k] = [1,2]
49  tdpbusd         tmm1, tmm4, tmm5
50  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1     // Store C tmp [m,n] = [1,0]
51  tdpbusd         tmm3, tmm4, tmm6
52  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3     // Store C tmp [m,n] = [1,1]
53
54  vcvtdq2ps       zmm0 , [r11 + C_TMP_OFFSET(0,0) + 0*TILE_N_B]   // int32 -> float
55  vmovups         zmm1 , [r14 + Q_OFFSET(0)]                      // q-factors for N=0
56  vmovups         zmm2 , [r14 + BIAS_OFFSET(0)]                    // biases    for N=0
57  vfmadd213ps     zmm0 , zmm1 , zmm2                              // zmm0  = zmm0  * q + b
58  vcvtps2dq       zmm0 , zmm0                                     // float -> int32
59  vpxord          zmm3 , zmm3 , zmm3                              // Prepare zero ZMM
60  vpmaxsd         zmm0 , zmm0 , zmm3                              // RELU (int32)
61  vpmovusdb       [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0        // uint32 -> uint8
62  vcvtdq2ps       zmm4 , [r11 + C_TMP_OFFSET(0,0) + 4*TILE_N_B]   // int32 -> float
63  vfmadd213ps     zmm4 , zmm1 , zmm2                              // zmm4  = zmm4  * q + b
64  vcvtps2dq       zmm4 , zmm4                                     // float -> int32
65  vpmaxsd         zmm4 , zmm4 , zmm3                              // RELU (int32)
66  vpmovusdb       [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4        // uint32 -> uint8
67  vcvtdq2ps       zmm5 , [r11 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]   // int32 -> float
68  vfmadd213ps     zmm5 , zmm1 , zmm2                              // zmm5  = zmm5  * q + b
69  vcvtps2dq       zmm5 , zmm5                                     // float -> int32
70  vpmaxsd         zmm5 , zmm5 , zmm3                              // RELU (int32)
71  vpmovusdb       [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5        // uint32 -> uint8
```

Example 20-21. Convolution Code Fused with Post-Convolution Code (Contd.)

```
72  vcvtdq2ps       zmm6 , [r11 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]   // int32 -> float
73  vfmadd213ps     zmm6 , zmm1 , zmm2                              // zmm6  = zmm6  * q + b
74  vcvtps2dq       zmm6 , zmm6                                     // float -> int32
75  vpmaxsd         zmm6 , zmm6 , zmm3                              // RELU (int32)
76  vpmovusdb       [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6        // uint32 -> uint8
77  vcvtdq2ps     zmm7 , [r11 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]         // int32 -> float
78  vmovups       zmm8 , [r14 + Q_OFFSET(1)]                            // q-factors for N=1
79  vmovups       zmm9 , [r14 + BIAS_OFFSET(1)]                         // biases    for N=1
80  vfmadd213ps   zmm7 , zmm8 , zmm9                                    // zmm     7  = zmm7  * q + b
81  vcvtps2dq     zmm7 , zmm7                                           // float -> int32
82  vpmaxsd       zmm7 , zmm7 , zmm3                                    // RELU (int32)
83  vpmovusdb     [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7              // uint32 -> uint8
84  vcvtdq2ps     zmm10, [r11 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]         // int32 -> float
85  vfmadd213ps   zmm10, zmm8 , zmm9                                    // zmm10 = zmm10 * q + b
86  vcvtps2dq     zmm10, zmm10                                          // float -> int32
87  vpmaxsd       zmm10, zmm10, zmm3                                    // RELU (int32)
88  vpmovusdb     [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10             // uint32 -> uint8
89  vcvtdq2ps     zmm11, [r11 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]         // int32 -> float
90  vfmadd213ps   zmm11, zmm8 , zmm9                                    // zmm11 = zmm11 * q + b
91  vcvtps2dq     zmm11, zmm11                                          // float -> int32
92  vpmaxsd       zmm11, zmm11, zmm3                                    // RELU (int32)
93  vpmovusdb     [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11             // uint32 -> uint8
94  vcvtdq2ps     zmm12, [r11 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]         // int32 -> float
95  vfmadd213ps   zmm12, zmm8 , zmm9                                    // zmm12 = zmm12 * q + b
96  vcvtps2dq     zmm12, zmm12                                          // float -> int32
97  vpmaxsd       zmm12, zmm12, zmm3                                    // RELU (int32)
98  vpmovusdb     [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12             // uint32 -> uint8
```

20.11.2 INTEL® AMX AND INTEL® AVX-512 INTERLEAVING (SW PIPELINING)

A modern CPU has multiple functional units that can execute different instructions simultaneously. For example, a load instruction and an arithmetic instruction can execute in parallel. A commonly used approach for maximizing the utilization of various resources in parallel is the out-of-order execution, where the CPU might alter the order of the instructions to achieve higher resource utilization.

Intel AMX compute instructions are prime candidates for optimization because they utilize resources very lightly (1/2 of the available ALU ports, 1/TILE_M of the time).

Theoretically, the blue post-convolutional code of one iteration could execute in parallel to the Bold code in lines 3 through 25 (before the first TILESTORE) of the next iteration, where iteration is the execution of the code in Example 20-21. Unfortunately, this cannot be done automatically and efficiently by the CPU. Since the convolution (**Bold**) and post-convolution (blue) parts of the code tend to be sizable, the CPU can only overlap small portions efficiently before running out of resources in the out-of-order machine. Thus, a manual (SW) solution is required.

To reiterate, the blue code before the first TILESTORE can be run in parallel with the green code of the next iteration. This would overwrite temporary_C memory, which the post-convolution code reads from. To remove this dependency and maximize parallel execution, use double-buffering on temporary_C. Temporary_C would thus contain two buffers, interchanged every iteration.

In Example 20-32, the content deviates from the previous example by interleaving the current iteration's convolutional code with the previous iteration's post-convolutional code. Temporary_C is double-buffered, with r11 pointing to the buffer of the current iteration and r12 pointing to the previous iteration's buffer. They are exchanged at the end of the iteration.

Example 20-22. An Example of a Short GEMM Fused and Pipelined with Quantization and ReLU

```
1   ldtilecfg       tc                                          // Load tile config
2    mov            r15, 192                                    // A stride
3    mov            r13, 128                                    // B, C_TMP stride
4    TILELOADD      tmm5, [r9 + r13*1 + B_OFFSET(0,0)]          // Load B [k,n] = [0,0]
5    TILELOADD      tmm4, [r8 + r15*1 + A_OFFSET(0,0)]          // Load A [m,k] = [0,0]
6    TILEZERO       tmm0                                        // Zero acc [m,n] = [0,0]
7    vcvtdq2ps      zmm0, [r12 + C_TMP_OFFSET(0,0) + 0*TILE_N_B]  // int32 -> float
8    vmovups        zmm1, [r14 + Q_OFFSET(0)]                   // q-factors for N=0
9    vmovups        zmm2, [r14 + BIAS_OFFSET(0)]                // biases    for N=0
10   vfmadd213ps    zmm0, zmm1, zmm2                            // zmm0 = zmm0 * q + b
11   vcvtps2dq      zmm0, zmm0                                  // float -> int32
12   vpxord         zmm3, zmm3, zmm3                            // Prepare zero ZMM
13   vpmaxsd        zmm0, zmm0, zmm3                            // RELU (int32)
14   tdpbusd        tmm0, tmm4, tmm5
15  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(0,1)]          // Load B [k,n] = [0,1]
16  TILEZERO        tmm2                                        // Zero acc [m,n] = [0,1]
17  vpmovusdb       [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0    // uint32 -> uint8
18  vcvtdq2ps       zmm4 , [r12 + C_TMP_OFFSET(0,0) + 4*TILE_N_B]  // int32 -> float
19  vfmadd213ps     zmm4 , zmm1 , zmm2                          // zmm4  = zmm4  * q + b
20  tdpbusd tmm2, tmm4, tmm6
21  TILELOADD       tmm4, [r8 + r15*1 + A_OFFSET(1,0)]          // Load A [m,k] = [1,0]
22  TILEZERO        tmm1                                        // Zero acc [m,n] = [1,0]
23  vcvtps2dq       zmm4 , zmm4                                 // float -> int32
24  vpmaxsd         zmm4 , zmm4 , zmm3                          // RELU (int32)
25  vpmovusdb       [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4    // uint32 -> uint8
26  tdpbusd         tmm1, tmm4, tmm5
27  TILEZERO        tmm3                                        // Zero acc [m,n] = [1,1]
28  vcvtdq2ps       zmm5 , [r12 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]  // int32 -> float
29  vfmadd213ps     zmm5 , zmm1 , zmm2                          // zmm5  = zmm5  * q + b
30  vcvtps2dq       zmm5 , zmm5                                 // float -> int32
31  vpmaxsd         zmm5 , zmm5 , zmm3                          // RELU (int32)
32  tdpbusd         tmm3, tmm4, tmm6
33  TILELOADD       tmm5 , [r9 + r13*1 + B_OFFSET(1,0)]         // Load B [k,n] = [1,0]
34  TILELOADD       tmm4 , [r8 + r15*1 + A_OFFSET(0,1)]         // Load A [m,k] = [0,1]
35  vpmovusdb       [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5    // uint32 -> uint8
36  vcvtdq2ps       zmm6 , [r12 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]  // int32 -> float
37  vfmadd213ps     zmm6 , zmm1 , zmm2                          // zmm6  = zmm6  * q + b
38  tdpbusd         tmm0, tmm4, tmm5
39  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(1,1)]          // Load B [k,n] = [1,1]
40  vcvtps2dq       zmm6 , zmm6                                 // float -> int32
41  vpmaxsd               zmm6 , zmm6 , zmm3                    // RELU (int32)
42  vpmovusdb             [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6  // uint32 -> uint8
43  tdpbusd               tmm2 , tmm4, tmm6
```

Example 20-22. (Contd.)An Example of a Short GEMM Fused and Pipelined with Quantization and

```
44  TILELOADD       tmm4 , [r8 + r15*1 + A_OFFSET(1,1)]                 // Load A [m,k] = [1,1]
45  vcvtdq2ps       zmm7 , [r12 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]       // int32 -> float
46  vmovups         zmm8 , [r14 + Q_OFFSET(1)]                          // q-factors for N=1
47  vmovups         zmm9 , [r14 + BIAS_OFFSET(1)]                       // biases    for N=1
48  vfmadd213ps     zmm7 , zmm8 , zmm9                                  // zmm7  = zmm7  * q + b
49  vcvtps2dq       zmm7 , zmm7                                         // float -> int32
50  vpmaxsd         zmm7 , zmm7 , zmm3                                  // RELU (int32)
51  tdpbusd         tmm1 , tmm4, tmm5
52  vpmovusdb       [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7            // uint32 -> uint8
53  vcvtdq2ps       zmm10 , [r12 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]      // int32 -> float
54  vfmadd213ps     zmm10 , zmm8 , zmm9                                 // zmm10 = zmm10 * q + b
55  tdpbusd         tmm3 , tmm4, tmm6
56  TILELOADD       tmm5 , [r9 + r13*1 + B_OFFSET(2,0)]                 // Load B [k,n] = [2,0]
57  TILELOADD       tmm4 , [r8 + r15*1 + A_OFFSET(0,2)]                 // Load A [m,k] = [0,2]

58  vcvtps2dq       zmm10 , zmm10                                       // float -> int32
59  vpmaxsd         zmm10 , zmm10, zmm3                                 // RELU (int32)
60  vpmovusdb       [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10           // uint32 -> uint8
61  tdpbusd         tmm0, tmm4, tmm5
62  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0             // Store C tmp [m,n] = [0,0]
63  TILELOADD       tmm6, [r9 + r13*1 + B_OFFSET(2,1)]                  // Load B [k,n] = [2,1]
64  vcvtdq2ps       zmm11, [r12 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]       // int32 -> float
65  vfmadd213ps     zmm11, zmm8 , zmm9                                  // zmm11 = zmm11 * q + b
66  vcvtps2dq       zmm11, zmm11                                        // float -> int32
67  vpmaxsd         zmm11, zmm11, zmm3                                  // RELU (int32)
68  tdpbusd         tmm2, tmm4, tmm6
69  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2             // Store C tmp [m,n] = [0,1]
70  TILELOADD       tmm4, [r8 + r15*1 + A_OFFSET(1,2)]                  // Load A [m,k] = [1,2]
71  vpmovusdb       [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11           // uint32 -> uint8
72  vcvtdq2ps       zmm12, [r12 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]       // int32 -> float
73  vfmadd213ps     zmm12, zmm8 , zmm9                                  // zmm12 = zmm12 * q + b
74  tdpbusd         tmm1, tmm4, tmm5
75  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1             // Store C tmp [m,n] = [1,0]
76  vcvtps2dq       zmm12, zmm12                                        // float -> int32
77  vpmaxsd         zmm12, zmm12, zmm3                                  // RELU (int32)
78  vpmovusdb       [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12           // uint32 -> uint8
79  tdpbusd         tmm3, tmm4, tmm6
80  TILESTORED      [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3             // Store C tmp [m,n] = [1,1]
81
82  xchg            r11, r12                                            // Swap buffers for current/next iter
```

Except for a larger TILE_M (N_ACC=M_ACC=2, TILE_M=16, TILE_K=64, TILE_N=16) on a [256x192] x [192x256] GEMM, application of this algorithm with the parameters laid out in section Section 20.8.1 yielded an 18.5% improvement in running time vs. the non-interleaved code described in Section 20.11.1.

20.11.3 AVOIDING THE H/W OVERHEAD OF FREQUENT OPEN/CLOSE OPERATIONS IN PORT FIVE

When the processor executes Intel AMX compute instructions (TDP*), it usually closes port five (one of the two Intel AVX-512 FMA ports) to conserve power. When the processor senses no more Intel AMX compute instructions in the pipeline, it opens port five. This open/close operation stalls the pipeline for a few cycles. Up to 20% performance degradation may be observed when the Intel AVX-512 instruction block contains 100 to 300 Intel AVX-512 instructions.

We recommend adding one or two TILEZERO instructions in the middle of the green block, as illustrated in Figure 20-10, roughly one hundred Intel AVX-512 instructions apart. Such an addition ensures that port five remains closed during blocks of up to three hundred Intel AVX-512 instructions. For longer blocks, it is preferable not to insert TILEZERO since longer blocks execute faster on two open FMA ports. The processor does not open port five for blocks shorter than one hundred Intel AVX-512 instructions, so no special handling is necessary.

NOTE

The TILEZERO instruction is considered an Intel AMX compute instruction for that matter.

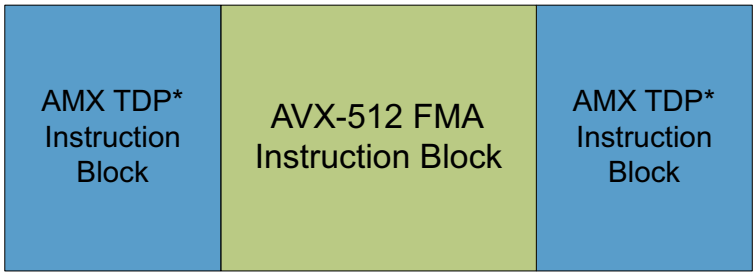


Figure 20-10. Using TILEZERO to Solve Performance Degradation

20.11.4 POST-CONVOLUTION MULTIPLE OFM ACCUMULATION AND EFFICIENT DOWN-CONVERSION

An important question arises concerning fused post-convolution optimization. What is the optimal block of accumulators processed in a single post-convolution iteration? As a post-processing unit, it is convenient to consider the M_ACC * N_ACC block of tiles accumulated in loops starting at lines 7-8 and 10-11 in Example 20-21 and Example 20-22, respectively. For simplicity, consider only multiples of these accumulation blocks. There is a trade-off between using smaller and larger post-convolution blocks:

Using small post-convolution blocks may have a negative impact by interrupting the convolution flow too often. Conversely, using big post-convolution blocks may also negatively impact by evicting part of the accumulated tiles out of DCU.

The optimal size, therefore, depends very much on the DL network topology and convolution-blocking parameters. Performance studies show that the number of iterations of M_ACC * N_ACC blocks before proceeding to post-convolution iteration may vary from 1 to 7.

As AMX instructions generate a higher precision output (32-bit integers or 32-bit floats) from lower precision inputs (8-bit integers or 16-bit bfloats, respectively), there is a need to convert 32-bit outputs to 8- or 16-bit inputs to be fed to the next DL network layer.

Suppose a single high-precision cache line (512-bit) is processed for conversion at a time. In that case, there will be two or four rounds of processing until a single low-precision cache line is generated for 8- or 16-bit inputs. Potential problems include:

- the number of loads and stores of the same cache line increases 4X or 2X, respectively.
- the next round of processing of the same cache line may occur after this cache line is evicted from DCU.

One of the optimizations mitigating these performance issues is to collect enough high-precision outputs to convert the full low-precision cache line in a single round.

The following drawing shows the conversion flow of 32-bit integers to 8-bit integers. Each colored block at the top represents a single **full** TILE output. The horizontal dimension is OFMs the vertical dimension is spatial).

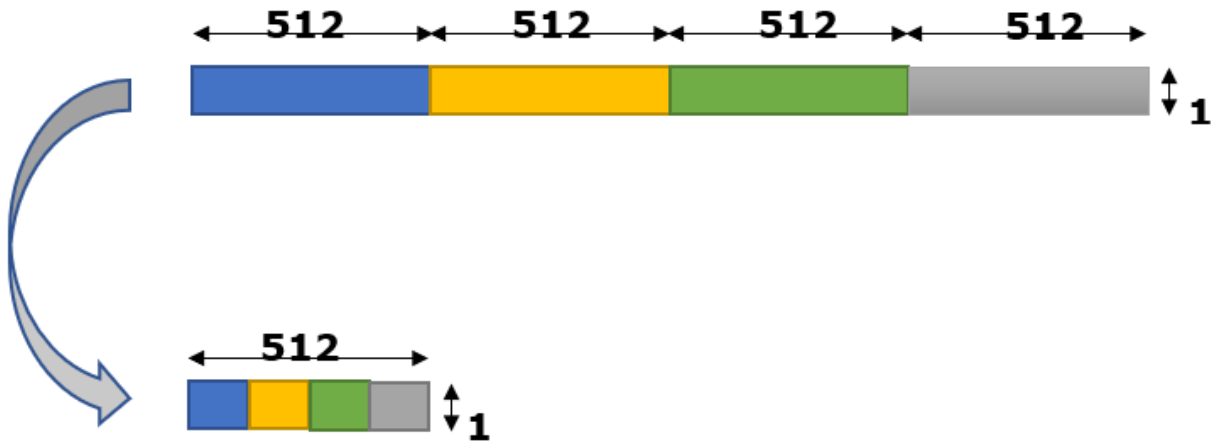


Figure 20-11. A Conversion Flow of 32-bit Integers to 8-bit Integers

To generate full 512-bit cache lines of 8-bit inputs (bottom), a multiple of 64 OFMs should be collected before conversion. Accordingly, to generate full cache lines with 16-bit inputs, a multiple of 32 OFMs should be collected. This often produces better performance results, though it may be viewed as a restriction to convolution-blocking parameters (in particular, N_ACC).

Example 20-23 shows the conversion code for two blocks of sixteen cache lines of 32-bit floats converted to a single block of sixteen cache lines of 16-bit bfloats. TMUL outputs are assumed to be placed into a scratchpad *spad,* and the conversion result is placed in the next_inputs buffer.

Example 20-23. Two Blocks of 16 Cache Lines of 32-bit Floats Converted to One Block of 16 Cache Lines of 16-bit BFloat

```
float* spad;
bfloat_16* next_inputs;
inline unsigned inputs_spatial_dim( void ) {
   return /* number of pixels in map */
}
for (int i = 0; i < 16; i++)
{
__m512 f32_0 = _mm512_load_ps(spad);
       __m512 f32_1 = _mm512_load_ps(spad + 16*16);
       __m512 bf16 = _mm512_castsi512_ps(_mm512_cvtne2ps_pbh(f32_1, f32_0));
       _mm512_store_ps(next_inputs, bf16);

     spad += 16; /* Next TILE row */
     next_inputs += 32 * inputs_spatial_dim();
}
```

Example 20-24. Using Unsigned Saturation

Example 1-10

```
const int32_t db_sel[16] = { 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15 };
inline __m512i Pack_DwordsToBytes(__m512i dwords[4])
{
   const __m512i sel_reg   = _mm512_load_si512(db_sel);
   const __m512i words_0  = _mm512_packs_epi32(dwords[0], dwords[1]);
   const __m512i words_1  = _mm512_packs_epi32(dwords[2], dwords[3]);
   __m512i bytes             = _mm512_packus_epi16(words_0, words_1);
   bytes                        = _mm512_permutexvar_epi32(sel_reg, bytes);

   return bytes;
}
```

20.12 INPUT AND OUTPUT BUFFERS REUSE (DOUBLE BUFFERING)

Due to the significant computational speedup achieved by the Intel AMX instructions, the performance bottleneck of Intel AMX-enabled applications is usually memory access. The most straightforward way to improve memory utilization is to reduce an application's memory footprint. An application with a smaller memory footprint will keep more of its essential data in the caches while reducing the number of costly cache evictions. This usually improves performance.

In Deep Learning (DL), a simple, efficient way to reduce the memory footprint is to reuse the input and output buffers of various layers in the topology.

Figure 20-12 illustrates where the previous layer feeds the next layer (left).

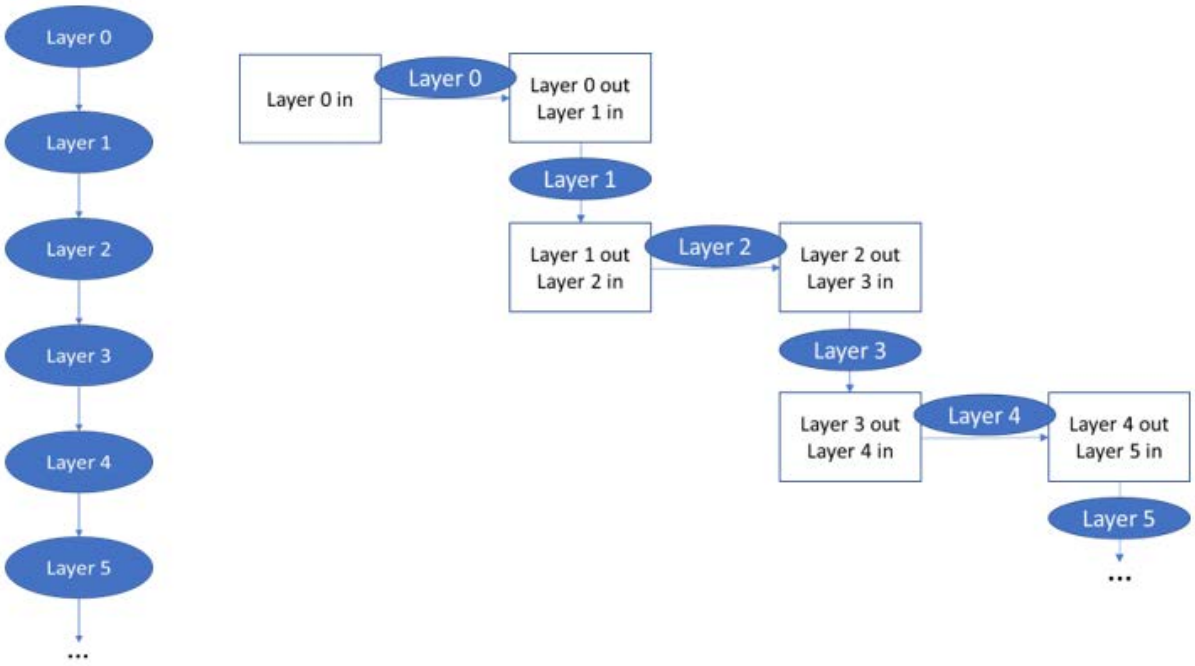


Figure 20-12. Trivial Deep Learning Topology with Naive Buffer Allocation

A straightforward buffer allocation scheme is illustrated in Figure 20-12, in which the output of layer N is placed into a dedicated memory buffer, which is then consumed as input by layer N+1. In this scheme, such topology with L-layers would require L+1 memory buffers, of which only the last is valuable (containing the final results). The rest of the L memory buffers are single-use and disposable,
significantly increasing the application's memory footprint.

The allocation scheme in Figure 20-13 offers an improved scheme whereby the entire topology only requires two reusable memory buffers.

Figure 20-13. Minimal Memory Footprint Buffer Allocation Scheme for Trivial Deep Learning Topology

A more complex topology would require more reusable buffers, but this number is significantly smaller than the naïve approach. ResNet-50, for example, requires only three reusable buffers (instead of 55). Inception-ResNet-V2 requires only five reusable buffers (instead of over 250). This optimization resulted in a 25% improved performance on the int8 end-to-end large batch throughput run of Resnet50 v1.5.

20.13 SOFTWARE PREFETCHES

The CPU employs sophisticated HW prefetchers that predict future access and provide relevant data. This works best when most memory accesses are sequential. For more details on processor hardware prefetchers, see Section 20.13.1.2.

20.13.1 SOFTWARE PREFETCH FOR CONVOLUTION AND GEMM LAYERS

Since the Conv/GEMM kernel is centered around loops over the M, K, and N dimensions of the involved matrices, the access will often be sequential. However, memory blocking, also recommended in this guide, causes the CPU to re-use the same block in the A or B matrices (or both) multiple times during the kernel execution. This means that sometimes the HW prefetcher cannot predict the subsequent access correctly. This opens the opportunity for an SW prefetch algorithm tightly integrated into the Conv/GEMM kernel and can bring in cache lines from future blocks based on the blocking strategy.

While the SW prefetch instruction enables selecting the target cache hierarchy level for the prefetch, this document assumes that the prefetch will go to the MLC. The DCU is too small to prevent the prefetched lines from being evicted before they can be used, and prefetching to LLC may not yield significant improvement.

20.13.1.1 The Prefetch Strategy

The prefetch strategy is highly dependent on the Conv/GEMM kernel method of operation. Assuming the “loops and blocking” design discussed earlier, the kernel operation can probably be split into multiple phases where each phase manages a different part of the matrices (corner, middle, etc.). The developer is encouraged to reduce the program’s size by reusing sections for repeatable matrix patterns to avoid overflowing the instruction cache. This can be done by having each section work on relative addresses. The SW prefetch instruction can be integrated into these sections and work on relative addresses. This means that while one section of the code loads addresses for its use, it also prefetches memory for a future section. The future section can be determined by looking at the future indices of any of the M/K/N loop levels.

20.13.1.2 Prefetch Distance

One of the most important decisions when using SW prefetching is the distance between the current and prefetched addresses. Supposing some blocking strategy is employed, it is more complex than adding an offset to the current address. The prefetched address must be set based on the target block of the matrix. If the target block is too close, the prefetched memory might still be in transit when the memory is required, and the CPU will stall, waiting for it to arrive. The data might be evicted if prefetched memory is too distant before it is used. The developer must tune the distance based on the layer/blocking
parameters.

As an example heuristic:

- One or two loads for each TMUL operation.
- Where one matrix is already in a register.
- When two registers must be loaded.
- The recommended range between the prefetch time and the consumption time is between 100 and 500 TMUL operations.
- 100 TMUL operations should take about 1600 cycles.
- The maximum number of bytes loaded between prefetch and consumption is 1MB (500 TMUL ops /* 2 loads per ops /* 1K per tile).

- The optimum is probably closer to 100 TMUL ops. At any rate, the developer must check the current CPU architecture and make sure that the MLC will not overflow.

20.13.1.3 To Prefetch A or Prefetch B?

Whether to prefetch A, B, or both depends on the order of layer execution.

In general, the following approaches are available:

- Image affinity.
- Execute the next layer of the same image.
- Complete a single image end-to-end before continuing to the next image in the same mini-batch.

Layer affinity:

- Execute the same layer of the following image.
- Complete a layer for all images in the mini-batch before continuing to the next layer.

The activations (the result of the previous layer) in the CPU caches are seen when image affinity is used. The weights in the caches are found when layer affinity is used. Generally, image affinity is recommended when sizeof(A)>sizeof(B) and layer affinity otherwise. To maximize performance, the developer should tune the switch point between the two methods. The choice between these two methods is also affected by the target matrix for prefetching.

Suppose the developer is confident that one of the matrices will already be present in the cache when the Conv/GEMM kernel begins execution. In that case, the potential benefit of SW prefetching decreases the potential benefit of SW prefetching decreases dramatically.

The size of the A-matrix, B-matrix, and cache.

The developer should sum up the memory requirements of the Conv/GEMM kernel for the current layer and compare it to the size of the cache (MLC). Combined with the previous step, it can indicate whether SW prefetching can yield any performance benefit. When large matrices are involved, there is a greater chance for improvement when prefetching the A- and the B-matrices.

20.13.1.4 To Prefetch or Not to Prefetch C?

It is not the C-matrix we might want to prefetch but rather the final output matrix of the layer after its post-convolution or post-GEMM phase, including quantization to a lower precision data type. Generally, prefetch those cache lines ahead of time since, with double buffering, these might have been read by previous layers, possibly executed in other cores.

Use the PREFETCHW instruction to read those cache lines into the DCU just in time for the store operations to find them in the DCU ready to be written, avoiding Read For Ownership latency that otherwise delays store completion. The exact timing of issuing the PREFETCHW instruction depends on multiple factors and requires careful tuning to get it right.

20.13.2 SOFTWARE PREFETCH FOR EMBEDDING LAYER

When the memory access pattern is semi-random, it is often impossible for the HW prefetcher to predict since it is based on application logic. In this case, the application may benefit from “proactive” prefetching using the SW prefetch instructions of addresses the application knows it will access soon.

An excellent example is Deep Learning, wherein the recommendation systems often use the embedding layer. The core loop of the embedding algorithm loads indices from an index buffer, and for each index, it loads the corresponding row from the embedding table for further processing. While the index buffer may contain duplicate

indices that benefit from CPU caching, the pattern is often considered random or semi-random. This can make the HW prefetcher less efficient. Since the entire content of the index buffer is already known, rows soon to be encountered can be prefetched to the DCU.

Example 20-25. Prefetching Rows to the DCU

```
1   void prefetched_embedding(uint32_t *a, float *e, float *c, size_t num_indices,
2                   float scale, float bias, size_t lookahead)
3   {
4           __m512 s = _mm512_set1_ps(scale);
5           __m512 b = _mm512_set1_ps(bias);
6
7           for (size_t i = 0; i < num_indices; i++) {
8                   _mm_prefetch(
9                           (char const *)&e[a[i + lookahead] * FLOATS_PER_CACHE_LINE],
10                          _MM_HINT_T0);
11                  __m512 ereg =
12                    _mm512_load_ps(&e[((size_t)a[i]) * FLOATS_PER_CACHE_LINE]);
13                  __m512 creg = _mm512_fmadd_ps(ereg, s, b);
14                  _mm512_store_ps(&c[i * FLOATS_PER_CACHE_LINE], creg);
15          }
16  }
```

20.14 HARDWARE PREFETCHER TUNING

L2P AMP prefetcher was introduced in the 5th Generation Intel® Xeon® Scalable processor based on the Emerald Rapids microarchitecture.

L2P auto-tunes the settings of the Adaptive Multipath Probability (AMP) prefetcher. In this case, the AMP prefetcher uses a machine learning algorithm to predict the best L2 prefetcher configuration for the currently running workload. This feature has been shown to improve the performance of various general-purpose workloads. However, on a few Intel AMX-based Deep Learning (DL) workloads, up to ~7% performance degradation was observed with the feature enabled.

For best DL workload performance, we recommend disabling the feature on 5th Generation Intel® Xeon® Scalable processors via Unified Extensible Firmware Interface UEFI/BIOS knob as follows:

1. Boot the machine into UEFI/BIOS.
2. Set knobs: Socket Configuration > Processor Configuration > AMP Prefetch: Disable
3. Save and reboot the machine.

20.15 STORE TO LOAD FORWARDING

Before it gets written to the DCU (first-level cache), store instructions copy data from general purpose, vector, or tile registers into store buffers. Besides TileLoad, all load instructions can load the data they seek from the store buffers and memory hierarchy.

The TileLoad instruction can’t load data from store buffers. It can only detect that the data is there and must wait until it is written to the memory hierarchy. Once written, TileLoad can read it from the memory hierarchy. This incurs a significant slowdown.

TILESTORE forwarding to non-TileLoad instructions via store buffers is supported under one restriction: they must both be of cache line size (64 bytes).

Forwarding is generally not advised because this mechanism has outliers. To avoid store-to-load forwarding, ensure enough distance between those two operations in the order of 10s of cycles in time.

20.16 MATRIX TRANSPOSE

This section describes the best-known SW implementations for several matrix transformations of BF16 data.

In this context, **flat format** means:

- Normal (i.e., non-VNNI).
- Unblocked rows (rows of matrices occupy a consecutive region in memory).

The first condition is essential. The second could be relaxed by changing the code in Example 20-26 accordingly. **VNNI format** implies only the second condition (non-blocking of rows). Notably, the MxN matrix in flat format will be represented by a (M/2)x(N/*2) matrix in VNNI format.

20.16.1 FLAT-TO-FLAT TRANSPOSE OF BF16 DATA

The primitive block transposed in this algorithm is 32x8 (i.e., 32 rows, eight BF16 numbers each), which is transformed into an 8x32 block (i.e., eight rows of 32 BF16 numbers each).

The implementation uses sixteen ZMM registers and three mask registers.

Input parameters: MxN, sizes of the rectangular block to be transposed. Assuming M is a multiple of 32, and N is a multiple of eight, we may also assume in Example 20-26:

- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains the starting address of the input matrix.
- r9 contains the starting address of the output buffer.

Example 20-26. (Sheet 1 of 3)BF16 Matrix Transpose (32x8 to 8x32)

```
/*1 of 3 */
1     mov         r10,          0xf0
2     kmovd       k1,           r10d
3     mov         r10,          0xf00
4     kmovd       k2,           r10d
5     mov         r10,          0xf000
6     kmovd       k3,           r10d
7     mov         rax,          N / 8
L.N:
8     mov         rdx,          M / 32
L.M:
9     vbroadcasti32x4 zmm0,                   xmmword ptr [r8]
10    vbroadcasti32x4 zmm0{k1},         xmmword ptr [r8+I_STRIDE*8]
11    vbroadcasti32x4 zmm0{k2},         xmmword ptr [r8+I_STRIDE*16]
12    vbroadcasti32x4 zmm0{k3},         xmmword ptr [r8+I_STRIDE*24]
13    vbroadcasti32x4 zmm1,             xmmword ptr [r8+I_STRIDE*1]
```

Example 20-26. (Contd.)(Sheet 2 of 3)BF16 Matrix Transpose (32x8 to 8x32)

```
11  vbroadcasti32x4 zmm0{k2},          xmmword ptr [r8+I_STRIDE*16]
12  vbroadcasti32x4 zmm0{k3},          xmmword ptr [r8+I_STRIDE*24]
13  vbroadcasti32x4 zmm1,              xmmword ptr [r8+I_STRIDE*1]
14  vbroadcasti32x4 zmm1{k1},          xmmword ptr [r8+I_STRIDE*9]
15  vbroadcasti32x4 zmm1{k2},          xmmword ptr [r8+I_STRIDE*17]
16  vbroadcasti32x4 zmm1{k3},          xmmword ptr [r8+I_STRIDE*25]
17  vbroadcasti32x4 zmm2,              xmmword ptr [r8+I_STRIDE*2]
18  vbroadcasti32x4 zmm2{k1},          xmmword ptr [r8+I_STRIDE*10]
19  vbroadcasti32x4 zmm2{k2},          xmmword ptr [r8+I_STRIDE*18]
20  vbroadcasti32x4 zmm2{k3},          xmmword ptr [r8+I_STRIDE*26]
21  vbroadcasti32x4 zmm3,              xmmword ptr [r8+I_STRIDE*3]
22  vbroadcasti32x4 zmm3{k1},          xmmword ptr [r8+I_STRIDE*11]
23  vbroadcasti32x4 zmm3{k2},          xmmword ptr [r8+I_STRIDE*19]
24  vbroadcasti32x4 zmm3{k3},          xmmword ptr [r8+I_STRIDE*27]
25  vbroadcasti32x4 zmm4,              xmmword ptr [r8+I_STRIDE*4]
26  vbroadcasti32x4 zmm4{k1},          xmmword ptr [r8+I_STRIDE*12]
27  vbroadcasti32x4 zmm4{k2},          xmmword ptr [r8+I_STRIDE*20]
28  vbroadcasti32x4 zmm4{k3},          xmmword ptr [r8+I_STRIDE*28]
29  vbroadcasti32x4 zmm5,              xmmword ptr [r8+I_STRIDE*5]
30  vbroadcasti32x4 zmm5{k1},          xmmword ptr [r8+I_STRIDE*13]
31  vbroadcasti32x4 zmm5{k2},          xmmword ptr [r8+I_STRIDE*21]
32  vbroadcasti32x4 zmm5{k3},          xmmword ptr [r8+I_STRIDE*29]
33  vbroadcasti32x4 zmm6,              xmmword ptr [r8+I_STRIDE*6]
34  vbroadcasti32x4 zmm6{k1},          xmmword ptr [r8+I_STRIDE*14]
35  vbroadcasti32x4 zmm6{k2},          xmmword ptr [r8+I_STRIDE*22]
36  vbroadcasti32x4 zmm6{k3},          xmmword ptr [r8+I_STRIDE*30]
37  vbroadcasti32x4 zmm7,              xmmword ptr [r8+I_STRIDE*7]
38  vbroadcasti32x4 zmm7{k1},          xmmword ptr [r8+I_STRIDE*15]
39  vbroadcasti32x4 zmm7{k2},          xmmword ptr [r8+I_STRIDE*23]
40  vbroadcasti32x4 zmm7{k3},          xmmword ptr [r8+I_STRIDE*31]
41  vpunpcklwd      zmm8,  zmm0,  zmm1
42  vpunpckhwd      zmm9,  zmm0,  zmm1
43  vpunpcklwd      zmm10, zmm2,  zmm3
44  vpunpckhwd      zmm11, zmm2,  zmm3
45  vpunpcklwd      zmm12, zmm4,  zmm5
46  vpunpckhwd      zmm13, zmm4,  zmm5
47  vpunpcklwd      zmm14, zmm6,  zmm7
48  vpunpckhwd      zmm15, zmm6,  zmm7
49  vpunpckldq      zmm0,  zmm8,  zmm10
50  vpunpckhdq      zmm1,  zmm8,  zmm10
51  vpunpckldq      zmm2,  zmm9,  zmm11
52  vpunpckhdq      zmm3,  zmm9,  zmm11
53  vpunpckldq      zmm4,  zmm12, zmm14
54  vpunpckhdq      zmm5,  zmm12, zmm14
55  vpunpckldq      zmm6,  zmm13, zmm15
56  vpunpckhdq      zmm7,  zmm13, zmm15
57  vpunpcklqdq     zmm8,  zmm0,  zmm4
58  vpunpckhqdq     zmm9,  zmm0,  zmm4
```

Example 20-26. (Contd.)(Sheet 3 of 3)BF16 Matrix Transpose (32x8 to 8x32)

```
59  vpunpcklqdq   zmm10,   zmm1,    zmm5
60  vpunpckhqdq   zmm11,   zmm1,    zmm5
61  vpunpcklqdq   zmm12,   zmm2,    zmm6
62  vpunpckhqdq   zmm13,   zmm2,    zmm6
63  vpunpcklqdq   zmm14,   zmm3,    zmm7
64  vpunpckhqdq   zmm15,   zmm3,    zmm7
65  vmovdqu16     zmmword ptr [r9],                    zmm8
66  vmovdqu16     zmmword ptr [r9+O_STRIDE],           zmm9
67  vmovdqu16     zmmword ptr [r9+O_STRIDE*2],         zmm10
68  vmovdqu16     zmmword ptr [r9+O_STRIDE*3],         zmm11
69  vmovdqu16     zmmword ptr [r9+O_STRIDE*4],         zmm12
70  vmovdqu16     zmmword ptr [r9+O_STRIDE*5],         zmm13
71  vmovdqu16     zmmword ptr [r9+O_STRIDE*6],         zmm14
72  vmovdqu16     zmmword ptr [r9+O_STRIDE*7],         zmm15
73  add       r9, 0x40
74  add       r8, I_STRIDE*32
75  dec       rdx
76  jnz       L.M
77  add       r9, (O_STRIDE*8 — (M/32) * 0X40)
78  sub       r8, (I_STRIDE*M-0x10)
79  dec       rax
80  jnz       L.N
```

Implementation discussion:

- Lines 1-6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9-72 implement the transpose of a primitive block 32x8. It uses 16 ZMM registers (zmm0-zmm15).

- Lines 9-40 implement loading 32 quarter-cache lines into 8 ZMM registers, according to Table 20-7 (numbers are in **bytes**):

Table 20-7. (Sheet 1 of 2)Loading 32 Quarter-Cache Lines into 8 ZMM Registers

		16	16	16	16	16
			ZIMM			
broadcast32x4	zmm0					
	zmm1					
	zmm2					
	zmm3					
	zmm4					
	zmm5					
	zmm6					
	zmm7					
broadcast32x4{0xf0}	**Zmm0**					
	zmm1					
	zmm2					
	zmm3					
	zmm4					
	zmm5					
	zmm6					
	zmm7					
broadcast32x4{0xf00}	zmm0					
	zmm1					
	zmm2					
	zmm3					
	zmm4					
	zmm5					
	zmm6					
	zmm7					

Table 20-7. (Contd.)(Sheet 2 of 2)Loading 32 Quarter-Cache Lines into 8 ZMM Registers

		16	16	16	16	16
			ZIMM			
broadcast32x4{0xf000}	zmm0					
	zmm1					
	zmm2					
	zmm3					
	zmm4					
	zmm5					
	zmm6					
	zmm7					

- Lines 41-64 are transpose flow proper. It creates a transposed block 8x32 in registers zmm8-zmm15.
- Lines 65-72 store transposed block 8x32 to the output buffer.

20.16.2 VNNI-TO-VNNI TRANSPOSE

The primitive block transposed in this algorithm is 8x8 (i.e., eight rows, eight BF16 numbers each), which is transformed into a 2x32 block (i.e., two rows of 32 BF16 numbers each).

The implementation uses five ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed (*in VNNI format*); it is assumed that M, N are multiples of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains the starting address to the input matrix.
- r9 contains the starting address to the output buffer.
- zmm31 is preloading with four copies of the following constant: unsigned int shuflle_cntrl[4] = {0x05040100, 0x07060302, 0x0d0c0908, 0x0f0e0b0a};

Example 20-27. (Sheet 1 of 2)BF16 VNNI-to-VNNI Transpose (8x8 to 2x32)

```
1       mov r10, 0xf0
2       kmovd k1, r10d
3       mov r10, 0xf00
4       kmovd k2, r10d
5       mov r10, 0xf000
6       kmovd k3, r10d
7       mov rax, N / 8
L.N:
8       mov rdx, M / 8
```

Example 20-27. (Contd.)(Sheet 2 of 2)BF16 VNNI-to-VNNI Transpose (8x8 to 2x32)

```
L.M:
9     vbroadcasti32x4 zmm0, xmmword ptr [r8]
10    vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*2]
11    vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*4]
12    vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*6]
13    vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
14    vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*3]
15    vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*5]
16    vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*7]
17    vpshufb         zmm2, zmm0, zmm31
18    vpshufb         zmm3, zmm1, zmm31
19    vpunpcklqdq     zmm0, zmm2, zmm3
20    vpunpckhqdq     zmm1, zmm2, zmm3
21    vmovdqu16 zmmword ptr [r9], zmm0
22    vmovdqu16 zmmword ptr [r9+O_STRIDE], zmm1
23    add r9, 0x40
24    add r8, I_STRIDE*8
25    dec rdx
26    jnz L.M
27    add r9, (O_STRIDE*2 - (M/8) * 0x40)
28    sub r8, (I_STRIDE*M-0x10)
29    dec rax
30    jnz L.N
```

BF16 VNNI-to-VNNI Transpose Implementation Discussion

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–22 implement the transpose of a primitive block 32x8. It uses five ZMM registers (zmm0-zmm3, zmm31).
- Lines 9–16 implement loading eight quarter-cache lines into two ZMM registers, according to Table 20-8 (numbers are in bytes).

- Lines 17–20 implement simultaneous transpose of four 2x2 blocks of QWORDs (i.e., 2x8 blocks of BF16). It

Table 20-8. Loading Eight Quarter-Cache Lines into Two ZMM Registers

		16	16	16	16	16
		ZIMM				
broadcast32x4	zmm0					
	zmm1					
broadcast32x4{0xf0}	zmm0					
	zmm1					
broadcast32x4{0xf00}	zmm0					
	zmm1					
broadcast32x4{0xf000}	zmm0					
	zmm1					

creates a transposed block 2x32 in registers zmm2-zmm3.

- Lines 21–22 store transposed block 2x32 to the output buffer.

20.16.3 FLAT-TO-VNNI TRANSPOSE

The algorithm below is based on: Flat-to-VNNI transpose of WORDs is equivalent to Flat-to-Flat transpose of DWORDs. This is illustrated below (the header numbers are bytes):

2	2	2	2
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Flat

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Flat transpose

2	2	2	2	2	2	2	2
0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15

VNNI transpose

Figure 20-14. Flat-to-VNNI Transpose of WORDs Equivalence to Flat-to-Flat Transpose of DWORDs

The primitive block transposed in this algorithm is 16x8 (i.e., 16 rows, 8 BF16 numbers each), which is transformed into a 4x32 block (i.e., four rows of 32 BF16 numbers each).

The implementation uses eight ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that M multiple of 16, N multiple of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains the starting address for the input matrix.
- r9 contains the starting address for the output buffer.

Example 20-28. (Sheet 1 of 2)BF16 Flat-to-VNNI Transpose (16x8 to 4x32)

```
 1    mov r10, 0xf0
 2    kmovd k1, r10d
 3    mov r10, 0xf00
 4    kmovd k2, r10d
 5    mov r10, 0xf000
 6    kmovd k3, r10d
 7    mov rax, N / 8
L.N:
 8    mov rdx, M / 16
L.M:
 9    vbroadcasti32x4 zmm0, xmmword ptr [r8]
10    vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*4]
11    vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*8]
12    vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*12]
13    vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
14    vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*5]
15    vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*9]
16    vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*13]
17    vbroadcasti32x4 zmm2, xmmword ptr [r8+I_STRIDE*2]
18    vbroadcasti32x4 zmm2{k1}, xmmword ptr [r8+I_STRIDE*6]
19    vbroadcasti32x4 zmm2{k2}, xmmword ptr [r8+I_STRIDE*10]
20    vbroadcasti32x4 zmm2{k3}, xmmword ptr [r8+I_STRIDE*14]
21    vbroadcasti32x4 zmm3, xmmword ptr [r8+I_STRIDE*3]
22    vbroadcasti32x4 zmm3{k1}, xmmword ptr [r8+I_STRIDE*7]
23    vbroadcasti32x4 zmm3{k2}, xmmword ptr [r8+ I_STRIDE*11]
24    vbroadcasti32x4 zmm3{k3}, xmmword ptr [r8+ I_STRIDE*15]
25    vpunpckldq      zmm4, zmm0, zmm1
26    vpunpckhdq      zmm5, zmm0, zmm1
27    vpunpckldq      zmm6, zmm2, zmm3
28    vpunpckhdq      zmm7, zmm2, zmm3
29    vpunpcklqdq     zmm0, zmm4, zmm6
30    vpunpckhqdq     zmm1, zmm4, zmm6
31    vpunpcklqdq     zmm2, zmm5, zmm7
32    vpunpckhqdq     zmm3, zmm5, zmm7
33    vmovups         zmmword ptr [r9], zmm0
34    vmovups         zmmword ptr [r9+O_STRIDE], zmm1
```

Example 20-28. (Contd.)(Sheet 2 of 2)BF16 Flat-to-VNNI Transpose (16x8 to 4x32)

```
35  vmovups             zmmword ptr [r9+O_STRIDE*2], zmm2
36  vmovups             zmmword ptr [r9+O_STRIDE*3], zmm3
37  add r9, 0x40
38  add r8, I_STRIDE*16
39  dec rdx
40  jnz L.M
41  add r9, (O_STRIDE*4 - (M/16)*0x40)
42  sub r8, (I_STRIDE*M-0x10)
43  dec rax
44  jnz L.N
```

Implementation Discussion:

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–36 implement the transpose of a primitive block 16x8. It uses eight ZMM registers (zmm0–zmm7).
- Lines 9–24 implement loading 16 quarter-cache lines into four ZMM registers zmm0-zmm3, according to Table 20-9 (numbers are in bytes):

Table 20-9. (Sheet 1 of 2)BF16 Flat-to-VNNI Transpose

		16	16	16	16
		ZIMM			
broadcast32x4	zmm0				
	zmm1				
	zmm2				
	zmm3				
broadcast32x4{0xf0}	**zmm0**				
	zmm1				
	zmm2				
	zmm3				
broadcast32x4{0xf00}	zmm0				
	zmm1				
	zmm2				
	zmm3				
broadcast32x4{0xf000}	zmm0				
	zmm1				
	zmm2				

Table 20-9. (Contd.)(Sheet 2 of 2)BF16 Flat-to-VNNI Transpose

		16	16	16	16
	zmm3				

- Lines 25–32 are the transpose flow proper. It creates a transposed block 4x32 in registers zmm0–zmm3.
- Lines 33–36 store transposed block 4x32 to the output buffer.

20.16.4 FLAT-TO-VNNI RE-LAYOUT

The primitive block which is being re-layout in this algorithm is 2x32 (i.e., 2 rows, 32 BF16 numbers each), which is transformed into a 1x64 block (i.e., 1 rows of 64 BF16 numbers each).

The implementation uses **5 ZMM registers and no mask registers**.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that **M multiple of 2, N multiple of 32.**
- I_STRIDE is the row size of input matrix in **bytes**.
- O_STRIDE is the row size of output buffer in **bytes**.
- r8 contains the starting address to the input matrix.
- r9 contains the starting address to the output buffer.
- zmm30, zmm31 are preloaded with following constants, respectively:
  - const short perm_cntl_1[32] = {0x00, 0x20, 0x01, 0x21, 0x02, 0x22, 0x03, 0x23, 0x04, 0x24, 0x05, 0x25, 0x06, 0x26, 0x07, 0x27, 0x08, 0x28, 0x09, 0x29, 0x0a, 0x2a, 0x0b, 0x2b, 0x0c, 0x2c, 0x0d, 0x2d, 0x0e, 0x2e, 0x0f, 0x2f};
  - const short perm_cntl_2[32] = {0x30, 0x10, 0x31, 0x11, 0x32, 0x12, 0x33, 0x13, 0x34, 0x14, 0x35, 0x15, 0x36, 0x16, 0x37, 0x17, 0x38, 0x18, 0x39, 0x19, 0x3a, 0x1a, 0x3b, 0x1b, 0x3c, 0x1c, 0x3d, 0x1d, 0x3e, 0x1e, 0x3f, 0x1f};

Example 20-29. (Sheet 1 of 2)BF16 Flat-to-VNNI Re-Layout

```
1     mov rdx, M / 2
L.M:
2     mov rax, N / 32
L.N:
3     vmovups zmm0, zmmword ptr [r8]
4     vmovups zmm1, zmmword ptr [r8+I_STRIDE]
5     vmovups zmm2, zmm0
6     vpermt2w zmm2, zmm30, zmm1
7     vpermt2w zmm1, zmm31, zmm0
8     vmovups zmmword ptr [r9], zmm2
```

Example 20-29. (Contd.)(Sheet 2 of 2)BF16 Flat-to-VNNI Re-Layout

```
9     vmovups zmmword ptr [r9+0x40], zmm1
10    add r9, 0x80
11    add r8, 0x40
12    dec rax
13    jnz L.N
14    add r9, (O_STRIDE - (N/32)*0x80)
15    add r8, (I_STRIDE*2 – (N/32)*0x40)
16    dec rdx
17    jnz L.M
```

Implementation Discussion:

- Lines 1 and 2 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 3 and 4 implement loading two full cache lines into two ZMM registers, zmm0-zmm1, from consecutive rows of the input matrix.
- Lines 5—7 implement the re-layout of a primitive block 2x32. It uses five ZMM registers (zmm0–zmm2, zmm30-zmm31).
- Lines 8 and 9 implement storing two full cache lines in two ZMM registers, zmm1-zmm2, into consecutive columns of the output matrix.

20.17 MULTI-THREADING CONSIDERATIONS

20.17.1 THREAD AFFINITY

As with Intel AVX-512 code, it is advised to fully define thread affinity and object affinity to process a single object in the same physical core, thus keeping the activations in core caches (unless larger than the size of the caches). This advice is imperative with Intel AMX code since those applications are more sensitive to memory-related issues.

20.17.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

Running more than one Intel AMX thread on the same physical core may result in overall performance loss due to the two threads competing for the same hardware resources. Scheduling a non-Intel AMX thread next to an Intel AMX thread on the same core may decrease the thread performance more than one expects due to normal competition for resources.

For optimum performance, please choose one of the following options in priority order:

1. Schedule one Intel AMX thread per physical core on one of its logical processors, while leaving the other logical processors idle.
2. Affintize a software thread that executes an endless TPAUSE CO.2 loop next to the Intel AMX thread.
   a. This prevents other threads from being scheduled on that logical processor.
      1) All hardware resources within the physical core are therefore allocated to the Intel AMX thread.
      2) This endless loop thread must terminate when the Intel AMX thread is about to terminate.
3. Code pause loops of thread pool threads waiting for the next task to be assigned to them with UMWAIT or TPAUSE C0.2 rather than with PAUSE, TPAUSE C0.1, or a non-pausing spin.

20.17.3 WORK PARTITIONING BETWEEN CORES

Deep Learning (DL) applications must often adhere to latency requirements that cannot be fulfilled within a single core. In these cases, a single object's processing must be partitioned between multiple cores.

Additionally, one layer's output is often the next layer's input. Due to the nature of the computations in DL applications, partitioning over different dimensions (N, M, K) will have different implications for the data locality in the core's caches. Minimize importing data from a different core's caches if possible, as this can hamper performance.

20.17.3.1 Partitioning Over M

Partitioning a DL layer over the M dimension has the advantage of nearly complete data locality. The layer's output is also partitioned by M between the cores and is, therefore, already in the cache of the corresponding core at the beginning of the next layer. Figure 20-15 shows this schematically.

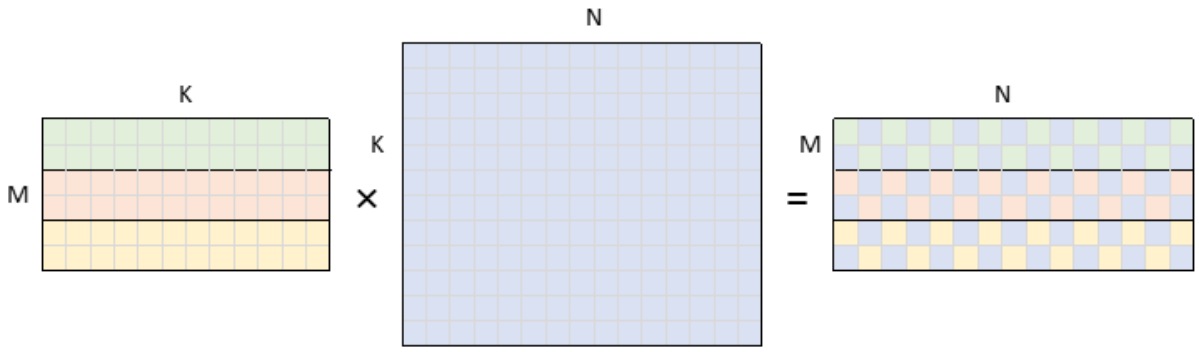


Figure 20-15. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the M-Dimension

Here the data read and written by each of the three cores is bound by a black rectangle.
It should be noted that in the case of convolutions, limited overlap in the M-dimension of the activations occurs between neighboring cores. Due to the convolutions, a finite-sized filter is slid over the activations. Thus, the M-dimension overlaps (KH-1)/*W (refer to Example 20-30) between the two neighboring cores.

- **Advantages:** When multiple layers in a chain are partitioned by the M-dimension between the same number of cores, each core has its data in its local cache.
- **Disadvantages:** All the cores read the B-matrix (or weights in convolutions) entirely, which might pose a bandwidth problem if the B-matrix is large.

20.17.3.2 Partitioning Over N

Partitioning a DL layer over the N-dimension reduces the read bandwidth in GEMMs with large B-matrices or large weights in convolutions. Each core reads a portion of the B-matrix in this scenario:

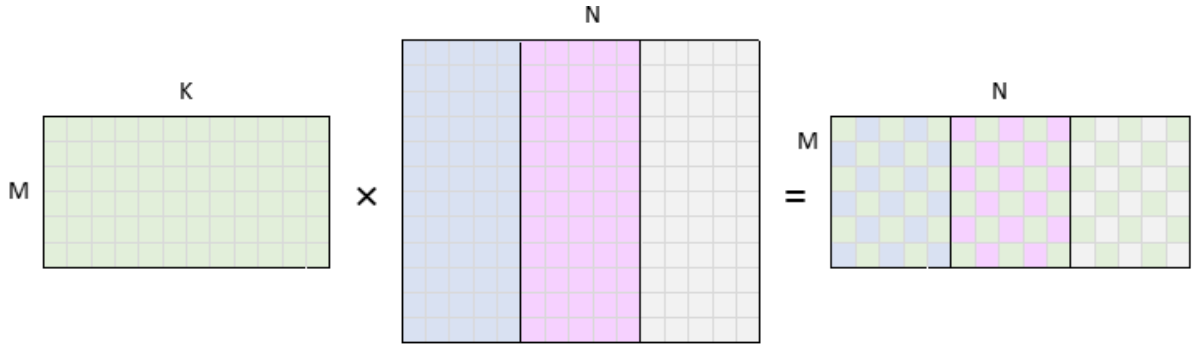


Figure 20-16. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the N-Dimension

Unfortunately, the layer's output is also partitioned by the N-dimension between the cores, which is incompatible with the M and N partitioning of the subsequent layer. For visualization, compare the right side of Figure 20-16 to the left side of Figure 20-15 and 20-16. In this scenario, a core in the subsequent layer is guaranteed to have most of its data from outside its local caches. This is not the case in K-dimension partitioning (see Section 20.17.3.3), but it also comes at a price.

- **Advantages:** It may reduce read bandwidth significantly in case of large B / large weights.
- **Disadvantages:** If the next layer is partitioned by M or N, most of the activations in the next layer will not reside in the local caches of the corresponding cores.

20.17.3.3 Partitioning Over K

Partitioning a DL layer over the K-dimension reduces the read bandwidth in GEMMs with large K-dimensions by reducing the amount of data being read from the A- and B-matrices (activations and weights in convolutions). Each core reads a portion of the matrices in this scenario, as illustrated in Figure 20-17.

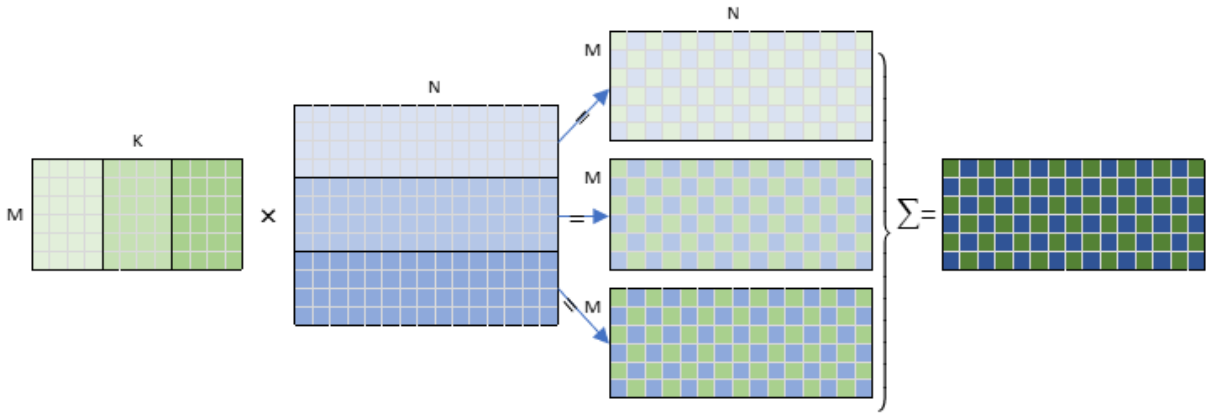


Figure 20-17. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the K-Dimension

Suppose a layer is partitioned by the N-dimension, and the K-dimension partitions the subsequent layer. In that case, the activation data will reside in the local caches of the cores in the layer partitioned by the K-dimension. For

visualization, compare the right side of Figure 20-16 with the left side of Figure 20-17. Unfortunately, this comes at a price: each core prepares partial results of the entire C-matrix.

To obtain final results, either a mutex (or several mutexes) is required to guard the write operations into the C-matrix, or a reduction operation is needed at the end of the layer. The mutex solution is not advised because threads will be blocked for a significant time. A reduction runs the risk of being costly since it entails the following:

- A synchronization barrier is required before the reduction.
- Reading a potentially large amount of data during the reduction:
  - There are T copies of the C-matrix, where T is the number of threads (the example has three).
  - The size of the matrices before the reduction is x2 (in case of a bfloat16 datatype) or x4 (in case of int8 datatype) times larger than the output C-matrix.
  - During the reduction, most of the cores' data will come outside their local cache hierarchy.

20.17.3.4 Memory Bandwidth Implications of Work Partitioning Over Multiple Dimensions

OpenMP offers a convenient interface for nested loop parallelization. For example, one could partition the N, M, and K dimensions can be partitioned automatically between threads using Example 20-30.

Example 20-30. GEMM Parallelized with omp Parallel for Collapse

```
#pragma omp parallel for collapse(2)
for (int n = 0; n < N; n += N_ACC*TILE_N) {
      for (int m = 0; m < M; m += M_ACC*TILE_M) {
  ...
 }
}
```

The collapse clause specifies how many loops within a nested loop should be collapsed into a single iteration space and divided between the threads. The order of the iterations in the collapsed iteration space is the same as though they were executed sequentially.

OpenMP automatically uses schedule(static,1) if there is no specified schedule, resulting in the sequential assignment of loop iterations to threads.

If we assume N=4*N_ACC*TILE_N and M=4*M_ACC*TILE_M wherein the K-dimension is deliberately excluded from consideration due to its problematic nature, there would be 4*4=16 iterations in the two nested loops. Now assume the division of iterations between three threads. Table 20-10 shows that the code in Example 20-30 would partition the iterations between threads.

Table 20-10. A Simple Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.3	1.2	2.1	3.0	3.3	100%	100%	38%
Thread 1:	0.1	1.0	1.3	2.2	3.1		100%	100%	100%
Thread 2:	0.2	1.1	2.0	2.3	3.2		100%	100%	100%

Every cell of the form n',m' contains the n'=n/N_ACC*TILE_N and m'=m/M_ACC*TILE_M indices from the loops in Example 20-30.

It is clear from Table 20-10 that each of the three threads executes at least one iteration with n'=0,1,2,3 and at least one iteration with m'=0,1,2,3. This means that every thread reads all of A and B.

By rearranging the work between threads in the following partitioning, the size of the B read is reduced by each thread by 50%, which might be significant in workloads where B is large. Similarly, the size of A can be reduced by 50% by swapping m' and n' indices for workloads with a large A.

Table 20-11. An Optimized Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.1	0.2	0.3	3.0	3.1	100%	50%	38%
Thread 1:	1.0	1.1	1.2	1.3	3.2	3.3	100%	50%	38%
Thread 2:	2.0	2.1	2.2	2.3			100%	25%	25%

20.17.4 RECOMMENDATION SYSTEM EXAMPLE

Many recommendation systems are built from a few GEMM layers that follow each other, an Embedding layer, and a layer connecting them. They are generally split into four distinct tasks:

1. Bottom GEMMs (MLPs).
2. Embedding.
3. Bottom MLP + Embedding Concat, GEMM, and Reshape.
4. Top GEMMs (MLPs).

The first two are independent so that they can execute in parallel. Their output feeds into the third task, whose output, in turn, feeds into the fourth task.

A few notes:

- Recommendation systems usually use a large batch to rank a reasonably large set of options.
- The GEMM layers are usually compute- or cache-bandwidth limited, whereas the Embedding layer is memory-bandwidth limited.
- Recommendation systems are real-time and thus limited to a specific latency.

When the latency requirement is a few milliseconds, the recommendation system topology has to be multi-threaded across several cores. The previous section discussed GEMM partitioning across multiple cores. This section deals with work partition between the four different tasks.

Figure 20-18 proposes a method of splitting the three tasks across machine cores. The block sizes in the chart are for illustration purposes only and do not represent any specific recommendation system.

Those three tasks can be split into two due to Bottom MLPs and Embedding independence.

- Those two tasks feed the other tasks:
  - Bottom MLP + Embedding Concat
    - **GEMM,**
    - **Reshape**
    - **Top MLPs.**
- The latter tasks are merged into a single task.
  - Choosing the number of cores for each task is a trial-and-error exercise.
    - **It may involve a phase for analyzing time required to execute each task across different cores.**

Because of a dependency between the bottom MLPs, embedding tasks, and the third task, a barrier exists between them, implying a potential wait time immediately following the faster layers.

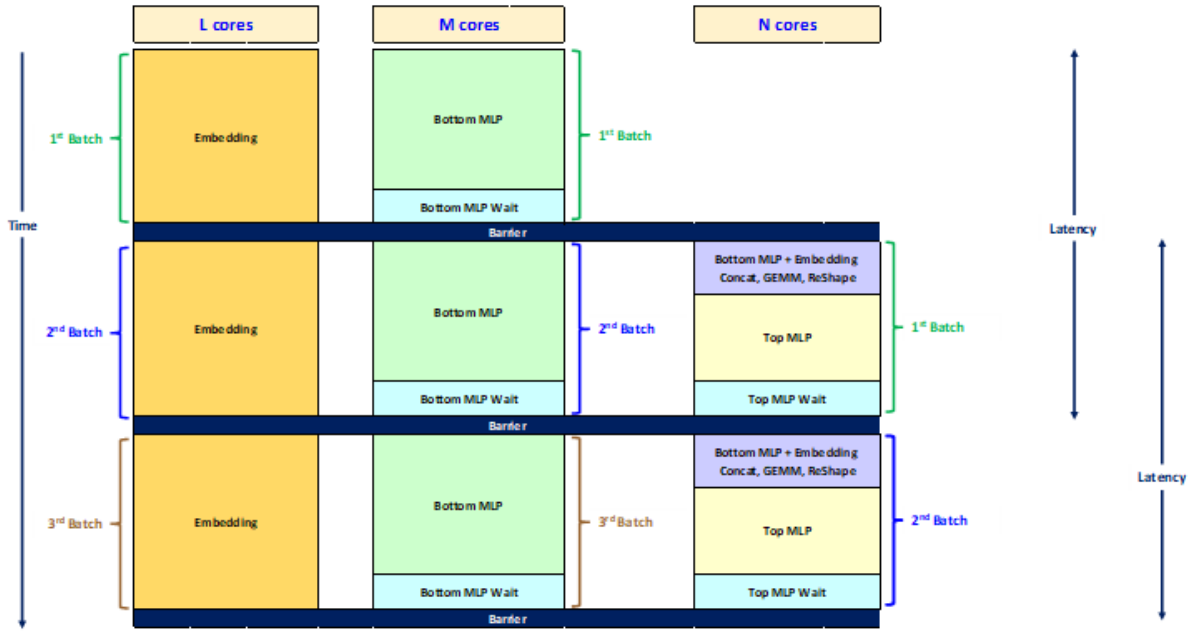


Figure 20-18. A Recommendation System Multi-Threading Model

20.18 SPARSITY OPTIMIZATIONS FOR INTEL® AMX

This section describes how Intel AMX can be further optimized for operations on sparse matrices. An example use case can be the inference of sparse neural networks, where the sparse weights are known to initially reside in DRAM due to the "online" usage model or large model capacity. In those cases, the primary performance bottleneck would be bringing the weights from DRAM. A helpful optimization
technique for this case is to get the weights from DRAM in a compressed format, decompress them into the local caches using Intel AVX-512, and perform Intel AMX computations on the decompressed data.

The compressed matrix format can consist of the following components:

- **compressed[]:** an array of non-zero matrix entries.
- **mask[]:** a bit-per-element array for the full matrix. 0 signifies the corresponding element is 0. 1 signifies a non-zero value in the **compressed[]** array mentioned above.

The compressed format can be computed off-line. The sparsity bitmask **mask[]** can be generated using the Intel AVX-512 *VPTESTMB* instruction on the sparse data. The **compressed[]** array can be generated using the Intel AVX-512 *VPCOMPRESS* instruction on the sparse data using the sparsity bitmask.

The code in Example 20-31 uses Intel AVX-512 to generate ***num*** rows of decompressed data, assuming 8-bit elements and 64 elements per tile row.

Example 20-31. Byte Decompression Code with Intel® AVX-512 Intrinsics

```
// uint8_t* compressed_ptr is a pointer to compressed data array
// __mmask64* compression_masks_ptr is a pointer to bitmask array
// uint8_t* decompressed_ptr is a pointer to decompressed data array

for (int i=0; i < num ; i++) {
  __m512i compressed = _mm512_loadu_epi32(compressed_ptr);
  __mmask64 mask = _load_mask64(compression_masks_ptr);
  __m512i decompressed_vec = _mm512_maskz_expand_epi8(mask, compressed);
  _mm512_store_epi32(decompressed_ptr, decompressed_vec);
  decompressed_ptr += 64; // 64 bytes per decompressed row
  compressed_ptr += _mm_countbits_64(mask); // advance compressed pointer by number of non-zero elements
  compression_masks_ptr ++; //64 bitmask bits per decompressed row
}
```

The matrix multiplication code will load the decompressed matrix to tiles from **decompressed[]**, an array containing the decompressed matrix data.

The decompression code uses of the Intel AVX-512 date expand operation as shown in Figure 20-19.

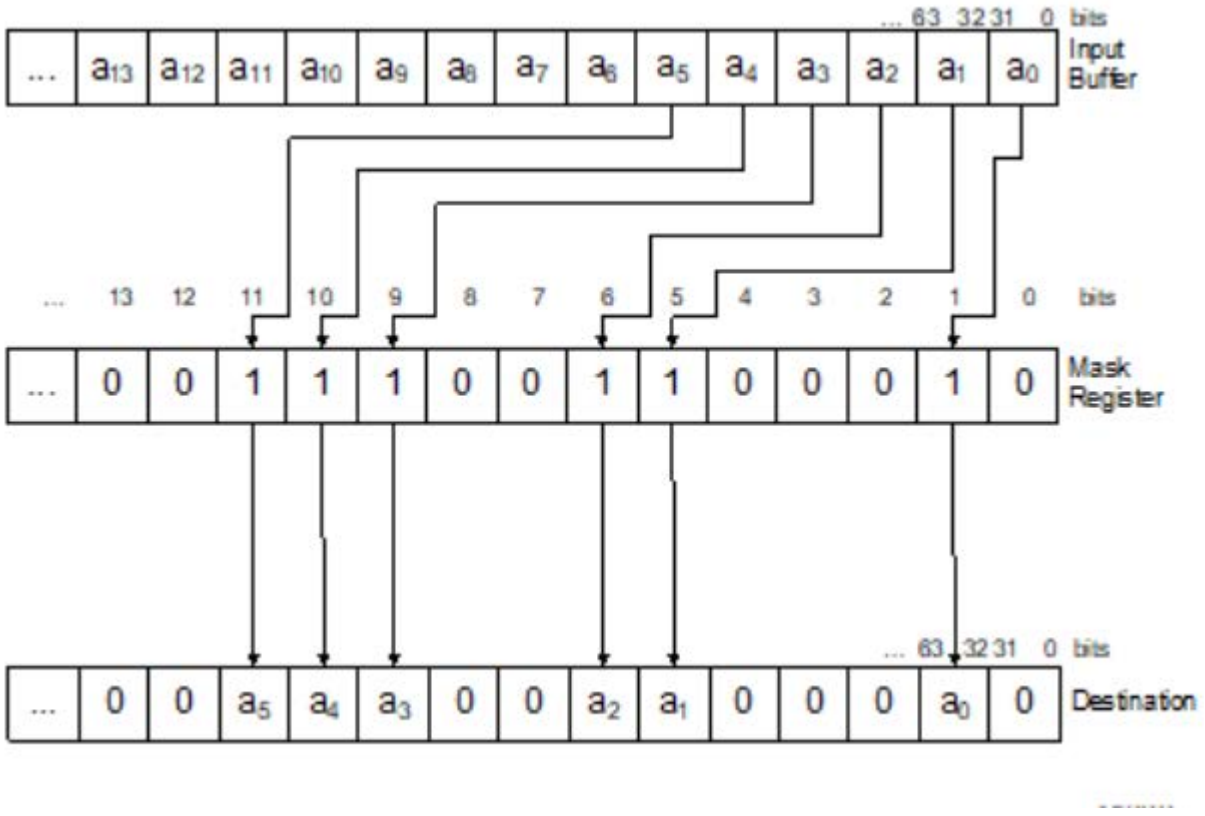


Figure 20-19. Data Expand Operation

Decompression code for 16-byte elements can be designed in the same way.

For the best performance, apply the following optimizations:

- **Interleaving:** Fine-grained interleaving of decompression code and matrix multiplication to overlap Intel AVX-512 decompression with Intel AMX computation.
- **Decompress Early:** Before immediate Intel AMX use, prepare the decompressed buffer to avoid store forwarding issues.
- **Buffer Reuse:** Decompressing the full sparse matrix could overflow the CPU caches. For best cache reuse, it is recommended to have a decompressed buffer that can hold two decompressed panels of the sparse matrix. While the matrix is multiplying with one panel, decompress the next panel for the subsequent iteration. In the subsequent iteration, decompress the next panel into the first half of the decompressed buffer that is no longer used, and so on.
- **Decompress Once:** Coordinate the matrix multiplication blocking and loop structure with the decompression scheme to minimize the number of times the same portion of the sparse matrix is decompressed. For example, if the B-matrix is sparse, traversing the entire vertical M-dimension will compress every vertical panel of the B-matrix only once.

20.19 TILECONFIG/TILERELEASE, CORE C-STATE, AND COMPILER ABI

For a function to use tile registers, it needs to configure them. For the LDTILECFG instruction definition, see Section 20.2. LDTILECFG creates an Intel AMX state which is kept valid until the TILERELEASE instruction is issued. TILERELEASE resets the Intel AMX state back to INIT. When the Intel AMX state is valid, and the OS issues the MWAIT instruction trying to move the physical processor, it executes on to Core C6 State. The 4th Generation Intel® Xeon® Scalable processor based on the Sapphire Rapids microarchitecture will not enter Core C6 even if the sibling logical processor is idle. This is because it lacks the dedicated backing store to keep the Intel AMX state until waking up. The Core C-State is demoted to C1 instead.

This is not an issue in Linux and Windows, where only the idle process issues the MWAIT instruction. The Idle Process in both operating systems does not use the Intel AMX ISA, so its Intel AMX tile state is always invalid (INIT). If still valid, the Intel AMX tile state will have previously been saved in an
OS-defined area in memory while context-switching between a thread that uses Intel AMX and the Idle Process thread.

20.19.1 ABI

The tile data registers (tmm0 – tmm7) are volatile. Their contents are passed back and forth between functions through memory. No tile register is saved and restored by the callee. Tile configuration is also volatile. The compiler saves and restores tile configuration and tile register contents if the register(s) need to live across the function call. The compiler eliminates the save instruction because its content remains the same on the stack. It reuses the configured content saved on the stack before the call. All functions must configure the tile registers themselves; however, tile registers may not be configured across functions.

Please download the System V Application Binary Interface: Intel386 Architecture Processor Supplement, Version1.0.

20.19.2 INTRINSICS

Example 20-32. Identification of Tile Shape Using Parameter m, n, k

```
typedef int _tile1024i __attribute__((__vector_size__(1024), __aligned__(64)));
_tile1024i _tile_loadd_internal(unsigned short m, unsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_loaddt1_internal(unsigned short m, uunsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_dpbssd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbsud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbusd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbuud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbf16ps_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
void_tile_stored_internal(unsigned short m, unsigned short n, void*base, __SIZE_TYPE__ stride, _tile1024i tile);
```

The parameter m, n, k identifies the shape of the tile.

20.19.3 USER INTERFACE

Example 20-33. Intel® AMX Intrinsics Header File

```
/* 1 of 4*/
typedef struct __tile1024i_str {
  const unsigned short row;
const unsigned short col;
  _tile1024i tile;
} __tile1024i;

/// Load tile rows from memory specified by "base" address and "stride" into destination tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILELOADD </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_loadd(__tile1024i *dst, const void *base, __SIZE_TYPE__ stride);
/// Load tile rows from memory specified by "base" address and "stride" into destination tile "dst".
///This intrinsic provides a hint to the implementation that the data will likely not be reused in the near future and
///the data caching can be optimized accordingly.
/// \headerfile <immintrin.h>
///
```

Example 20-33. (Contd.)Intel® AMX Intrinsics Header File

```
/// This intrinsic corresponds to the <c> TILELOADDT1 </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_stream_loadd(__tile1024i* dst, const void* base, __SIZE_TYPE__ stride);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with corresponding signed 8-bit integers in src1,
/// producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBSSD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbssd(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with corresponding unsigned 8-bit integers in src1,
/// producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer
/// in "dst", and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBSUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
void __tile_dpbf16ps(__tile1024i* dst, __tile1024i src0, __tile1024i src1);
/// Store the tile specified by "src" to memory specified by "base" address and "stride".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILESTORED </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
```

Example 20-33. (Contd.)Intel® AMX Intrinsics Header File

```
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be stored in memory.
void __tile_stored(void *base, __SIZE_TYPE__ stride, __tile1024i src);
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbsud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of unsigned 8-bit integers in src0 with corresponding signed 8-bit integers in
src1,
/// producing 4 intermediate 32-bit results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBUUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbuud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Zero the tile specified by "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILEZERO </c> instruction.
///
/// \param dst
///    The destination tile to be zero. Max size is 1024 Bytes.
void __tile_zero(__tile1024i* dst);
/// Compute dot-product of BF16 (16-bit) floating-point pairs in tiles src0 and src1, accumulating the intermediate
single-
/// precision (32-bit) floating-point elements with elements in "dst", and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBF16PS </c> instruction.
////// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
```

Example 20-33. (Contd.)Intel® AMX Intrinsics Header File

```
void __tile_dpbf16ps(__tile1024i* dst, __tile1024i src0, __tile1024i src1);
/// Store the tile specified by "src" to memory specified by "base" address and "stride".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILESTORED </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be stored in memory.
void __tile_stored(void *base, __SIZE_TYPE__ stride, __tile1024i src);
```

20.19.4 INTEL® AMX INTRINSICS EXAMPLE

In Example 20-34, function foo is called in line 18, and the tile variable ‘a’ written in line 17 needs to live up to line 21 across the function call. The compiler needs to save the tile data register allocated to ‘a’ before calling foo, then restore the tile configure register and tile data registers after calling foo. Lines 39, 42, and 46 in Example 20-35 are the save/restore code. Since the configure register doesn’t change, the configure register in the stack does not require saving.

Example 20-34. Intel® AMX Intrinsics Usage

```
 1 #include <immintrin.h>
 2
 3 char buf[1024];
 4 #define STRIDE 32
 5
 6 int count = 0;
 7 __attribute__((noinline))
 8 void foo() {
 9   count++;
10 }
11
12 void test_api(int cond, unsigned short row, unsigned short col) {
13   __tile1024i a = {row, col};
14   __tile1024i b = {row, col};
15   __tile1024i c = {row, col};
16
17   __tile_loadd(&a, buf, STRIDE);
18   foo();
19   __tile_loadd(&b, buf, STRIDE);
20   __tile_loadd(&c, buf, STRIDE);
21   __tile_dpbssd(&c, a, b);
22   __tile_stored(buf, STRIDE, c);
23 }
```

clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables.

Notice the ldtilecfg instruction at the beginning of the function (line 34 in Example 20-35), which sets the Intel AMX registers configuration within the CPU and the TILERELEASE instruction towards the end of the function. This placement ensures that the Intel AMX state is initialized, thus avoiding the expensive Intel AMX state save/restore in case of a software thread context-switch outside the Intel AMX function.

Example 20-35. Compiler-Generated Assembly-Level Code from Example 20-30

```
16 test_api:                    # @test_api
17 # %bb.0:                      # %entry
18      pushq   %rbp
19      pushq   %r15
20      pushq   %r14
21      pushq   %rbx
22      subq    $1096, %rsp             # imm = 0x448
23      movl    %edx, %ebx
24      movl    %esi, %ebp
25      vpxord  %zmm0, %zmm0, %zmm0
26      vmovdqu64       %zmm0, (%rsp)
27      movb    $1, (%rsp)
28      movw    %bx, 20(%rsp)
29      movb    %bpl, 50(%rsp)
30      movw    %bx, 18(%rsp)
31      movb    %bpl, 49(%rsp)
32      movw    %bx, 16(%rsp)
33      movb    %bpl, 48(%rsp)
34      ldtilecfg       (%rsp)
35      movl    $buf, %r14d
36      movl    $32, %r15d
37      TILELOADD       (%r14,%r15), %tmm0
38      movabsq $64, %rax
39      TILESTORED      %tmm0, 64(%rsp,%rax)    # 1024-byte Folded Spill
40      vzeroupper
41      callq   foo
42      ldtilecfg       (%rsp)
43      TILELOADD       (%r14,%r15), %tmm0
44      TILELOADD       (%r14,%r15), %tmm1
45      movabsq $64, %rax
46      TILELOADD       64(%rsp,%rax), %tmm2    # 1024-byte Folded Reload
47      tdpbssd %tmm0, %tmm2, %tmm1
48      TILESTORED      %tmm1, (%r14,%r15)
49      addq    $1096, %rsp             # imm = 0x448
50      popq    %rbx
51      popq    %r14
52      popq    %r15
53      popq    %rbp
54      TILERELEASE
55      retq
```

20.19.5 COMPILATION OPTION

The save/restore is sometimes unnecessary, e.g., when foo does not clobber any tile register. To avoid unnecessary save/restore, compile with **-mllvm -enable-ipra**, which does an IPO analysis to get the information on what physical registers are clobbered during the function call. Example 20-36 shows no tile register save/restore across calling foo.

clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables -mllvm -enable-ipra

Example 20-36. Compiler-Generated Assembly-Level Code Where Tile Register Save/Restore is Optimized Away

```
15      .type   test_api,@function
16  test_api:                           # @test_api
17 # %bb.0:                            # %entry
18      subq    $72, %rsp
19      vpxord  %zmm0, %zmm0, %zmm0
20      vmovdqu64       %zmm0, 8(%rsp)
21      movb    $1, 8(%rsp)
22      movw    %dx, 28(%rsp)
23      movb    %sil, 58(%rsp)
24      movw    %dx, 26(%rsp)
25      movb    %sil, 57(%rsp)
26      movw    %dx, 24(%rsp)
27      movb    %sil, 56(%rsp)
28      ldtilecfg       8(%rsp)
29      movl    $buf, %eax
30      movl    $32, %ecx
31      TILELOADD       (%rax,%rcx), %tmm0
32      callq   foo
33      TILELOADD       (%rax,%rcx), %tmm1
34      TILELOADD       (%rax,%rcx), %tmm2
35      tdpbssd %tmm1, %tmm0, %tmm2
36      TILESTORED      %tmm2, (%rax,%rcx)
37      addq    $72, %rsp
38      TILERELEASE
39      vzeroupper
40      retq
41 .Lfunc_end1:
42      .size   test_api, .Lfunc_end1-test_api
```

20.20 INTEL® AMX STATE MANAGEMENT

Intel AMX is XSAVE supported, meaning that it defines processor registers that can be saved and restored using instructions of the XSAVE feature set. Intel AMX is also XSAVE enabled, meaning that system software must enable it before it can be used.

The XSAVE feature set operates on state components, each a discrete set of processor registers (or parts of registers). Intel AMX is associated with two state components, XTILECFG and XTILEDATA. The XSAVE feature set organizes state components using state-component bitmaps. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. Intel AMX defines bits 18:17 for its state components (collectively, these are called AMX state):

- State component 17 is used for the 64-byte TILECFG register (XTILECFG state).
- State component 18 is used for the 8192 bytes of tile data (XTILEDATA state).

These are both user-state components, meaning the entire XSAVE feature set can manage them. In addition, it implies that setting bits 18:17 of extended control register XCR0 by system software enables Intel AMX. If those bits are zero, an Intel AMX instruction execution results in an invalid opcode exception (#UD).

About the XSAVE feature set's INIT optimization, the Intel AMX state is in its initial configuration if the TILECFG register is zero and all tile data are zero.

Enumeration and feature-enabling documentation can be found in Section 20.2.

An execution of XRSTOR or XRSTORS initializes the TILECFG register (resulting in TILES_CONFIGURED = 0) in response to an attempt to load it with an illegal value. Moreover, an execution of XRSTOR or XRSTORS that is not directed to load XTILEDATA leaves it unmodified, even if the execution is loading XTILECFG.

It is highly recommended that developers execute TILERELEASE to initialize the tiles at the end of the Intel AMX instructions code region. More on this is in Section 20.19.

If the system software does not initialize the Intel AMX state first (by executing TILERELEASE, for example), it may disable Intel AMX by clearing XCR0[18:17], by clearing CR4.OSXSAVE, or by setting IA32_XFD[18].

20.20.1 EXTENDED FEATURE DISABLE (XFD)

The XTILEDATA state component size is 8 KBytes, and an operating system may, by default, prefer not to allocate memory for the XTILEDATA state for every user thread. An operating system that enables Intel AMX might select a fault when user threads use the feature. That way, it can allocate a large enough state save area only for the user threads using the feature. An operating system may offer an API for the user threads to declare their intention to use Intel AMX and allow the OS to preallocate the state and avoid an exception when Intel AMX is used for the first time.

See Linux API and Windows API for more details.

Extended feature disable (XFD) is added to the XSAVE feature set to support such usage. See the Intel® AMX Architecture Definition for XFD documentation.

20.20.2 ALTERNATE SIGNAL HANDLER STACK IN LINUX OPERATING SYSTEM

When programs use an alternate signal handler stack, the stack size should be adjusted to accommodate the additional Intel AMX state. See Using XSTATE Features in User-Space Applications for more details.

20.21 USING INTEL® AMX TO EMULATE HIGHER PRECISION GEMMS

Intel AMX/TMUL has instructions that enable matrix-matrix operations such as multiplication on small precision elements. This section considers how to use the low-precision Intel AMX instructions to approximate the answers to matrix-matrix multiplication of higher-precision terms. Even if low-precision inputs are Bfloat16 or Integer8, one can still combine the transforms to approximate matrix-matrix multiplication in higher precisions.

Pay attention to the exponent range and mantissa bits when approximating higher precisions. There are IEEE-754 double precision numbers (FP64) that aren't representable as single precision (FP32) or lower precisions. These are typically range-based issues in the exponent bits. FP64 has more exponent bits than FP32. However, scaling factors can overcome most range-based problems. If A is a matrix of FP64 values, then A (as a sum of Bfloat16 matrices) cannot generally be represented. Scaling factors can, however, be used to get around most issues. The A-matrix as

s1*A1 + s2*A2 + … + sn*An can be written where each matrix A_i is lower precision, and each si is a constant scaling factor.

For Bfloat16 decomposition of FP32, consider the following:

- Let A be a matrix of FP32 values.
- Let A1 = bfloat16(A), a matrix containing RNE-rounded Bfloat16 conversions of A.
- Let A2 = bfloat16(A – fp32(A1)).
- Let A3 = bfloat16(A – fp32(A1) – fp32(A2)).
- Now A is approximately A1 + A2 + A3.

Once one has written two matrices as a sum of lower precision matrices, one can run AMX/TMUL on the product to approximate the higher precision. But to do this effectively, one needs to have higher precision accumulation. There are tricks in the literature for doing higher precision all in a lower precision, such as works on so-called double-double arithmetic. Still, these tend to vary too much from standard
matrix-matrix multiplication to be helpful with TMUL. In the case of Bfloat16, having 32-bit accumulation in the product allows one to use Bfloat16 to approximate FP32 accuracy.

Therefore, if A = s1*A1 + s2*A2 + s3*A3, and B = t1*B1 + t2*B2 + t3*B3, then A*B can be computed using AMX/TMUL on the projects Ai*Bj for 1<=i,j<=3, assuming scaling is done carefully to avoid
denormals. Assuming FP32 accumulation, the FP32 approximation of A*B can be made by writing out these lower precision multiplies. Scaling factors can be chosen to avoid denormals at times, but they can also be picked in a way that simplifies further steps in the algorithm. In some cases, scaling factors can be chosen to be a power of two, for instance, without significantly reducing the accuracy of the resulting matrix-matrix multiply.

The number of matrices for A or B are picked depending on the mantissa range to cover. If trying to emulate FP32 which has 24 bits of mantissa (including the implicit mantissa bit), it is possible with three Bfloat16 matrices (because each of the triples has 8 bits of mantissa, including the implicit bit.). Here, the range is less important because Bfloat16 and FP32 have the same exponent range. Use three Bfloat16 matrices to approximate FP32 precision by BF16x3. Range issues may still come up for BF16x3 cases where A has values close to the maximum or minimum exponent for FP32, but that too can be circumvented by scaling constants. Scaling factors of 2^24 or 2^(-24) suffice to push it far enough away from the boundary to make the computation feasible again. This is dependent upon the closest end of the spectrum.

A few terms from an expansion can also be dropped. For instance, in the BF16x3 case, where there are three As and three Bs, nine products may result. That is:

A*B = (A1+A2+A3)*(B1+B2+B3) = (A1*B1) + (A1*B2 + A2*B1) + (A1*B3 + A2*B2 + A3*B1) + (A2*B3 + A3*B2) +(A3*B3).

The parentheses in the last equation are intentionally derived so that all entries in the same “bin” are put together, and there are nine entries of the form Ai*Bj. This example has five bins, each with its own set of parentheses. In the Bfloat16 case, |Ai| <= |A_i-1}| / 256. This shows the last two bins (with A2*B3,A3*B2,A3*B3) are too small to contribute significantly to the answer, which is why if there are Y terms on each side of A*B, only (Y+1)*Y/2 multiplies are required, not Y*Y multiplies. In this case,
dropping the last three (also the difference between Y*Y – (Y+1)*Y/2 when Y=3.) from the nine
multiplies. The last three multiplies in the last two bins have terms less than 2^(-24) as big as the first term. So, A*B can be approximated (ignoring the scaling terms for now) as the sum of the first three most significant bins: A1*B1 + (A1*B2+A2*B1)+(A1*B3+A2*B2*A3*B1). In this case, adding from the least significant bin to the most significant bin (A1*B1) is recommended.

Whenever A and B are each expanded out to Y-terms, computing only Y*(Y+1)/2 products works under the condition that each term has the same number of mantissa bits. If some terms have a different number of bits, then this guideline no longer applies. But for BF16x3, each term covers eight mantissa bits and Y=3, so six products are needed.

Regarding accuracy, the worst-case relative error for BF16x3 may be worse than FP32. However, BF16x3 tends to cover a larger mantissa range due to implicit bits, which can be more accurate in many cases. Nevertheless, accuracy is not offered by matrix-matrix multiplication. Even FP64 or FP128 can be bad for component-wise relative errors. Take A = [1, -1] and B = [1; 1]. A*B is zero. Let eps be a small perturbation to A and/or B. The solution may now be arbitrarily bad in terms of relative error. In general, assume that the same mantissa range and exponent range is covered as a given higher-precision floating point format, and the accumulation is at least as good as the higher-precision format. With such an assumption, the answer will be approximately the same as the higher-precision floating point format. It may or may not be identical. Performing the same operation in the higher precision format but changing the order of the computations could yield slightly different results. In terms of matrix-matrix multiplication, it could yield vast differences in relative error.

Things get slightly more complicated if low precision is used to approximate matrix-matrix at FP64 accuracy or FP128 precision. Here the scalars aren't just for avoiding denormals but are necessary to do the initial matrix conversion. Nevertheless, converting to an integer is recommended in this case because the FP32-rounded errors in each of the seven or fewer bins may introduce too many errors. An integer is easier to get right because there are no floating-point errors in each bin.

Conversion to Integer functions in the same way as all of the previous Bfloat16 examples. The quantization literature explains how to map floating point numbers into integers. The only difference is that these integers are further broken down into 8-bit pieces for the use of Intel AMX. Constant factors are still needed, but in this case they are primarily defined in the conversion itself.

One difficulty with quantization to integers is the notion of a shared exponent. All the numbers quantized together with shared exponents must share the same range. The assumption is that all of A shares a joint exponent range. Since this will also be true for B, each row of A and column of B can be quantized separately.

Assuming that there is Integer32 accumulation with the Integer8 multiplies, a matrix may be broken down into far more bits than required. This may significantly reduce the inaccuracy impact of picking a shared exponent. Because Integer32 arithmetic will be precise, modulo overflow/underflow concerns, then one can break up A or B into a huge number of 8-bit integer matrices, then do all the matrix-matrix work with Intel AMX, and then convert back all the results to even get accuracies up to quad-precision.

Considering an extreme case of trying to get over 100-bits of accuracy in a matrix-matrix multiply.

All A-values can be quantified into 128-bit integers. The same holds true with B. Once broken down into 8-bit quantities, this will have a significant expansion like: A = s1*A1 + s2*A2 + … + s14*A14 for when attempting 112-bits of mantissa. The same can be done with B = t1*B1 + t2*B2 + … + t14*B14. A*B is potentially 14*14=196 products, but only 105 products are needed because the last few products may have scaling factors less than 2^(-112) times the most important terms. Each product term should be added separately and computing into C from the least significant bits forward.

C15 = (s1*t14)*A1*B14 + (s2*t13)*A2*B13 + … + (s14*t1)*A14*B1

- C14 = (s1*t13)*A1*B13 + (s2*t12)*A2*B12 + … + (s13*t1)*A13*B1
- C13 = (s1*t12)*A1*B12 + (s2*t11)*A2*B11 + … + (s12*t1)*A12*B1
- C02 = (s1*t1)*A1*B1

Sometimes choosing scalers is possible so all the products in a given row can be computed with the same scratch array. The converted sum of C02 gives the final product through C15, where terms like C15 should be computed first.

Writing matrix-matrix multiplies in terms of an expansion like (A1+A2+A3)*(B1+B2+B3) is referred to as "cascading GEMM." Performance will vary depending on the TMUL/Intel AMX specification, and may vary from generation to generation. Note that some computations may become bandwidth-bound. Since there is no quad floating-point

precision in hardware for Intel Architecture, the above algorithm may be competitive performance-wise with other approaches like doing software double-double optimizations or software-based quad precision.

CHAPTER 21
INTEL® QUICKASSIST TECHNOLOGY (INTEL® QAT)

The Intel® QuickAssist Technology (Intel® QAT) API supports two acceleration services:

- Cryptographic
- Data Compression.

The acceleration driver interfaces with the hardware via hardware-assisted rings. These rings are used as request and response rings. The driver uses request rings to submit requests to the accelerator and response rings to retrieve responses from the accelerator. The availability of responses can be indicated to the driver using either interrupts or by having software poll the response rings.

At the Intel QAT API, services are accessed via “instances.” A set of rings is assigned to an instance, and any operations performed on a service instance will involve communication over the rings assigned to that instance.

21.1 SOFTWARE DESIGN GUIDELINES

Key design decisions should be considered to achieve optimal performance when integrating with the Intel QuickAssist Technology software. In many cases, the best Intel® QuickAssist Technology configuration is dependent on the design of the application stack that is being used. Therefore, it is impossible to have a “one configuration fits all” approach. The trade-offs between different approaches will be discussed in this section to help the designer make informed decisions.

These guidelines focus on the following performance aspects:

- Maximizing throughput through the accelerator
- Minimizing the offload cost incurred when using the accelerator
- Minimizing latency

Each guideline will highlight its impact on performance. This document does not give specific performance numbers since exact performance numbers depend on various factors and tend to be specific to a given workload, software, and platform configuration. Further, such numbers tend to be specific to a given workload, software, and platform configuration.

21.1.1 POLLING VS. INTERRUPTS (IF SUPPORTED)

NOTE

Not all use cases support interrupt mode, and not all software packages support interrupt mode.

Software can either periodically query the hardware accelerator for responses or enable the generation of an interrupt when responses are available. Interrupts or polling mode can be configured per instance via the platform-specific configuration settings.

The properties and performance characteristics of each mode are explained below followed by recommendations on selecting a configuration.

21.1.1.1 Interrupt Mode

When operating in interrupt mode, the accelerator will generate an MSI-X interrupt when a response is placed on a response ring. Each ring bank has a separate MSI-X interrupt which may be steered to a particular CPU core via the CoreAffinity settings in the configuration file.

To reduce the number of interrupts generated, and hence the number of CPU cycles spent processing interrupts, multiple responses can coalesce together. The presence of the multiple responses can be indicated via a single coalesced interrupt rather than having an interrupt per response. An interrupt coalescing timer determines the number of responses associated with a coalesced interrupt. When the accelerator places a response in a response ring, it starts an interrupt coalescing timer. While the timer runs, additional responses may be placed in the response ring. When the timer expires, an interrupt is generated to indicate that responses are available.

Since interrupt coalescing is based on a timer, there is some variability in the number of responses associated with an interrupt. The arrival rate of responses is a function of the arrival rate of the associated requests and of the request size. Hence, the timer configuration needed to coalesce X large requests differs from the timer configuration needed to coalesce X small requests. Tuning the timer based on the average expected request size is recommended.

The choice of timer configuration impacts throughput, latency, and offload cost:

- Configuring a very short time period maximizes the throughput through the accelerator, minimizing latency, but will increase the offload cost since there will be a higher number of interrupts and hence more CPU cycles spent processing the interrupts.
  - If this interrupt processing becomes a performance bottleneck for the CPU, the overall system throughput will be impacted.
- Configuring a very long timer period leads to reduced offload cost (due to the reduction in the number of interrupts) but increased latency.
  - If the timer period is very long and causes the response rings to fill, the accelerator will stall, and throughput will be impacted.

The appropriate coalescing timer configuration will depend on the characteristics of the application. It is recommended that the value chosen is tuned to achieve optimal performance.

Using interrupts to notify user-space applications is achieved using “epoll mode,” which utilizes the kernel device drivers poll function to allow an application to get notified of interrupt events.

Because epoll mode has two parts, of which the kernel space part utilizes the interrupts, if there is a delay in the kernel interrupt (for example, by changing the coalescing fields), there will be a corresponding increase in latency in the delivery of the event to user space.

The thread waiting for an event in epoll mode does not consume CPU time, but the latency could impact performance. For higher packet load where the wait time for the next packet is insignificant, polling mode is recommended.

When using interrupts with the user space Intel QuickAssist Technology driver, there is significant overhead in propagating the interrupt to the user space process that the driver is running in. This leads to an increased offload cost. Hence, interrupts should not be used with the user-space Intel QuickAssist Technology driver.

21.1.1.2 Polling Mode

In polled mode, interrupts are fully disabled and the software application must periodically invoke the polling API, provided by the Intel® QuickAssist Technology driver, to check for and process responses from the hardware.

The frequency of polling is a key performance parameter that should be fine-tuned based on the application. This parameter has an impact on throughput, latency, and on offload cost:

- If the polling frequency is too high, CPU cycles are wasted if no responses are available when the polling routine is called. This leads to an increased offload cost.

- If the polling frequency is too low, latency is increased and throughput may be impacted if the response rings fill causing the accelerator to stall.

The choice of threading model has an impact on performance when using a polling approach. There are two main threading approaches when polling:

- Creating a polling thread that periodically calls the polling API. This model is often the simplest to implement, allows for maximum throughput, but can lead to increased offload cost due to the overhead associated with context switching to/from the polling thread.
- Invoking the polling API and submitting new requests from within the same thread. This model is characterized by having a "dispatch loop" that alternates between submitting new requests and polling for responses. Additional steps are often included in the loop such as checking for received network packets or transmitting network packets. This approach often leads to the best performance since the polling rate can be easily tuned to match the submission rate so throughput is maximized and offload cost is minimized.

21.1.1.3 Recommendations

Polling mode tends to be preferred in cases where traffic is steady (like packet processing applications) and can result in a minimal offload cost. Epoll mode is preferred for cases where traffic is bursty, as the application can sleep until there is a response to process.

Considerations when using polling mode:

- Fine-tuning the polling interval is critical to achieving optimal performance.
- The preference is for invoking the polling API and submitting new requests from within the same thread rather than having a separate polling thread.

Considerations when using epoll mode:

- CPU usage will be at 0% in idle state in epoll mode versus a non-zero value in standard poll mode. However, with a high load state, standard poll mode should out-perform epoll mode.

21.1.2 USE OF DATA PLANE (DP) API VS. TRADITIONAL API

The cryptographic and compression services provide two flavors of API, known as the traditional API and the Data Plane API. The traditional API provides a full set of functionality including thread safety that allows many application threads to submit operations to the same service instance. The Data Plane API is aimed at reducing offload cost by providing a "bare bones" API, with a set of constraints, which may suit many applications. Refer to the Intel QuickAssist Technology Cryptographic API Reference Manual for more details on the differences between the DP and traditional APIs for the crypto service.

From a throughput and latency perspective, there is no difference in performance between the Data Plane API and the traditional API.

From an offload cost perspective, the Data Plane API uses significantly fewer CPU cycles per request compared to the traditional API. For example, the cryptographic Data Plane API has an offload cost that is lower than the cryptographic traditional API.

21.1.2.1 Batch Submission of Requests Using the Data Plane API

The Data Plane API provides the capability to submit batches of requests to the accelerator. The use of the batch mode of operation leads to a reduction in the offload cost compared to submitting the requests one at a time to the accelerator. This is due to CPU cycle savings arising from fewer writes to the hardware ring registers in PCIe* memory space. However, it is important to note that optimized batch size may be different, depending on the application.

Using the Data Plane API, batches of requests can be submitted to the accelerator using either the cpaCySymDpEnqueueOp() or cpaCySymDpEnqueueOpBatch() API calls for the symmetric cryptographic data plane API and using either the cpaDcDpEnqueueOp() or

cpaDcDpEnqueueOpBatch() API calls for the compression data plane API. In all cases, requests are only submitted to the accelerator when the performOpNow parameter is set to CPA_TRUE.

It is recommended to use the batch submission mode of operation where possible to reduce offload cost.

21.1.3 SYNCHRONOUS (SYNC) VS. ASYNCHRONOUS (ASYNC)

The Intel QuickAssist Technology traditional API supports both synchronous and asynchronous modes of operation. The Intel QuickAssist Technology Data Plane API only supports the asynchronous mode of operation.

With synchronous mode, the traditional Intel QuickAssist Technology API will block and not return to the calling code until the acceleration operation has completed.

With asynchronous mode, the traditional or Data Plane Intel QuickAssist Technology API will return to the calling code once the request has been submitted to the accelerator. When the accelerator has completed the operation, the completion is signaled via the invocation of a callback function.

From a performance perspective, the accelerator requires multiple inflight requests per acceleration engine to achieve maximum throughput. With synchronous mode of operation, multiple threads must be used to ensure that multiple requests are inflight. The use of multiple threads introduces an overhead of context switching between the threads which leads to an increase in offload cost.

Hence, the use of asynchronous mode is the recommended approach for optimal performance.

21.1.4 BUFFER LISTS

The symmetric cryptographic and compression Intel QuickAssist Technology APIs use buffer lists for passing data to/from the hardware accelerator. The number and size of elements in a buffer list has an impact on throughput; performance degrades as the number of elements in a buffer list increases. To minimize this degradation in throughput performance, it is recommended to keep the number of buffers in a buffer list to a minimum. Using a single buffer in a buffer list leads to optimal performance.

NOTE

Specific performance numbers are not given in this document since exact performance numbers depend on a variety of factors and tend to be specific to a given workload, software and platform configuration.

When using the Data Plane API, it is possible to pass a flat buffer to the API instead of a buffer list. This is the most efficient usage of system resources (mainly PCIe bandwidth) and can lead to lower latencies compared to using buffer lists.

In summary, the recommendations for using buffer lists are:

- If using the Data Plane API, use a flat buffer instead of a buffer list.
- If using a buffer list, a single buffer per buffer list leads to highest throughput performance.
- If using a buffer list, keep the number of buffers in the list to a minimum.

21.1.5 MAXIMUM NUMBER OF CONCURRENT REQUESTS

The depth of the hardware rings used by the Intel QuickAssist Technology driver for submitting requests to, and retrieving responses from, the accelerator hardware can be controlled via the configuration file using the CyXNumConcurrentSymRequests, CyXNumConcurrentAsymRequests and DcXNumConcurrentRequests parameters. These settings can have an impact on performance:

- As the maximum number of concurrent requests is increased in the configuration file, the memory requirements required to support this also increases.
- If the number of concurrent requests is set too low, there may not be enough outstanding requests to keep the accelerator busy and throughput will degrade. The minimum number of concurrent requests required to keep the accelerator busy is a function of the size of the requests and of the rate at which responses are processed via either polling or interrupts (see Section 21.1.1 for additional details).
- If the number of concurrent requests is set too high, the maximum latency will increase.

It is recommended that the maximum number of concurrent requests is tuned to achieve the correct balance between memory usage, throughput and latency for a given application. As a guide the maximum number configured should reflect the peak request rate that the accelerator must handle.

21.1.6 SYMMETRIC CRYPTO PARTIAL OPERATIONS

The symmetric cryptographic Intel QuickAssist Technology API supports partial operations for some cryptographic algorithms. This allows a single payload to be processed in multiple fragments with each fragment corresponding to a partial operation. The Intel QuickAssist Technology API implementation will maintain sufficient state between each partial operation to allow a subsequent partial operation for the same session to continue from where the previous operation finished.

From a performance perspective, the cost of maintaining the state and the serialization between the partial requests in a session has a negative impact on offload cost and throughput. To maximize performance when using partial operations, multiple symmetric cryptographic sessions must be used to ensure that sufficient requests are provided to the hardware to keep it busy.

For optimal performance, it is recommended to avoid the use of partial requests if possible.

There are some situations where the use of partials cannot be avoided since the use of partials and the need to maintain state is inherent in the higher level protocol (such as, the use of the RC4 cipher with an SSL/TLS protocol stack).

21.1.7 REUSING SESSIONS IN QAT ENVIRONMENT

The session is the entry point to perform symmetric cryptography with the QAT device. Every session has assigned algorithm, state, instance, but also allocated memory space.

When limited the number of instances and want to run several different algorithms or change keys for another session, uninitialize the session and create a new one. However, such an approach impacts performance because it involves buffer disposal, deinitialization of the instance, etc..

Instead, the session can be reused with updating only a direction (encryption / decryption), key or symmetric algorithm to be used. This method will not dispose buffers and can reduce the CPU cycles significantly.

21.1.8 MAXIMIZING QAT DEVICE UTILIZATION

The Intel QuickAssist device utilization and throughput are maximized when there are sufficient requests outstanding to occupy the multiple internal acceleration engines with the device.

Assigning each Intel QuickAssist service instance to a separate CPU core to balance the load across the CPU is recommended to ensure that there are sufficient CPU cycles to drive the accelerators at maximum performance. In a CPU with sufficiently high frequency, multiple instances may share the same CPU core.

When using interrupts, the core affinity settings within the configuration file should be used to steer the interrupts for a service instance to the appropriate core.

21.1.9 BEST KNOWN METHOD (BKM) FOR AVOIDING PERFORMANCE BOTTLENECKS

For optimal performance, ensure the following:

- All data buffers should be aligned on a 64-byte boundary.
- Transfer sizes that are multiples of 64 bytes are optimal.
- Small data transfers (less than 64 bytes) should be avoided. If a small data transfer is needed, consider embedding this within a larger buffer so that the transfer size is a multiple of 64 bytes. Offsets can then be used to identify the region of interest within the larger buffer.
- Each buffer entry within a Scatter-Gather-List (SGL) should be a multiple of 64bytes and should be aligned on a 64-byte boundary.

21.1.10 AVOID DATA COPIES BY USING SVM AND ATS

On CPUs and Intel QuickAssist devices that support shared virtual memory (SVM), virtual addresses to virtually contiguous buffers can be supplied to the Intel QAT hardware. Without this support, physical addresses to physically contiguous and DMAable memory buffers must be used. Using virtual addressed memory avoids the need to copy payload data from user space memory allocated with malloc() to physically contiguous memory.

When SVM is enabled, the Intel QuickAssist device interacts with the IOMMU to fetch the virtual to physical address translations when accessing memory and this can result in increased latency and lower throughput.

21.1.11 AVOID PAGE FAULTS WHEN USING SVM

When using SVM to avoid data copies, there is a chance that after a request, that refers to a virtually addressed buffer, has been submitted to the Intel QuickAssist device, the operating system may swap out the memory pages associated with that buffer. This will result in a page fault when the Intel QAT device tries to access the memory. The Intel QAT device will stall the processing of that request until the page fault is resolved or times out. This can lead to an underutilization of the Intel QAT device. To avoid page faults, the memory submitted to QAT should be pinned.

CHAPTER 22
USING PERFORMANCE MONITORING EVENTS

Performance monitoring (Perfmon) provides means to characterize the interaction between programmed sequences of instructions and microarchitectural sub-systems. Performance monitoring facilities are described in Chapter 19, "Architectural Last Branch Records" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B. Performance-monitoring events are described at *https://perfmon-events.intel.com/*.

The first section of this appendix (Top-Down Analysis Method) provides information on the Top-Down Microarchitecture Analysis (TMA) method for analyzing performance bottlenecks when tuning for Intel microarchitectures. Sections 22.1.1 through 22.1.7 present a generalized formalism that can adapt to several recent Intel® microarchitectures.

The remaining subsections have instantiations of TMA for:

- Golden Cove microarchitecture
- Ice Lake microarchitecture
- Cascade Lake microarchitecture
- Skylake microarchitectures

The instantiations include examples where applicable.

The rest of this chapter has performance monitoring information for previous generations of Intel microarchitectures.

22.1 TOP-DOWN ANALYSIS METHOD

The section describes the Top-down Microarchitecture Analysis (TMA) method for identifying performance bottlenecks in out-of-order cores. The method's abstraction and the spirit of the hierarchical technique can apply to many out-of-order processors.

TMA simplifies cycle-accounting (the process of identifying costs of performance bottlenecks, also called CPI breakdown) using microarchitecture-abstracted metrics organized in one hierarchy.

General TMAMTMA Hierarchy for Out-of-Order Microarchitectures depicts the hierarchical approach to classify performance bottlenecks common to modern out-of-order microarchitectures. Using TMA, the high-learning curve associated with each microarchitecture generation is replaced by a structured drill-down that quickly guides the user to true performance limiters. This enables analyzing performance without requiring knowledge of every detail of the microarchitecture.

The advantage of this top-down hierarchical framework is a structured approach to drill down and guide you toward the likely area of microarchitecture to investigate. Weights are assigned to nodes in the tree to enable a focus analysis efforts on issues that matter and disregard minor issues.

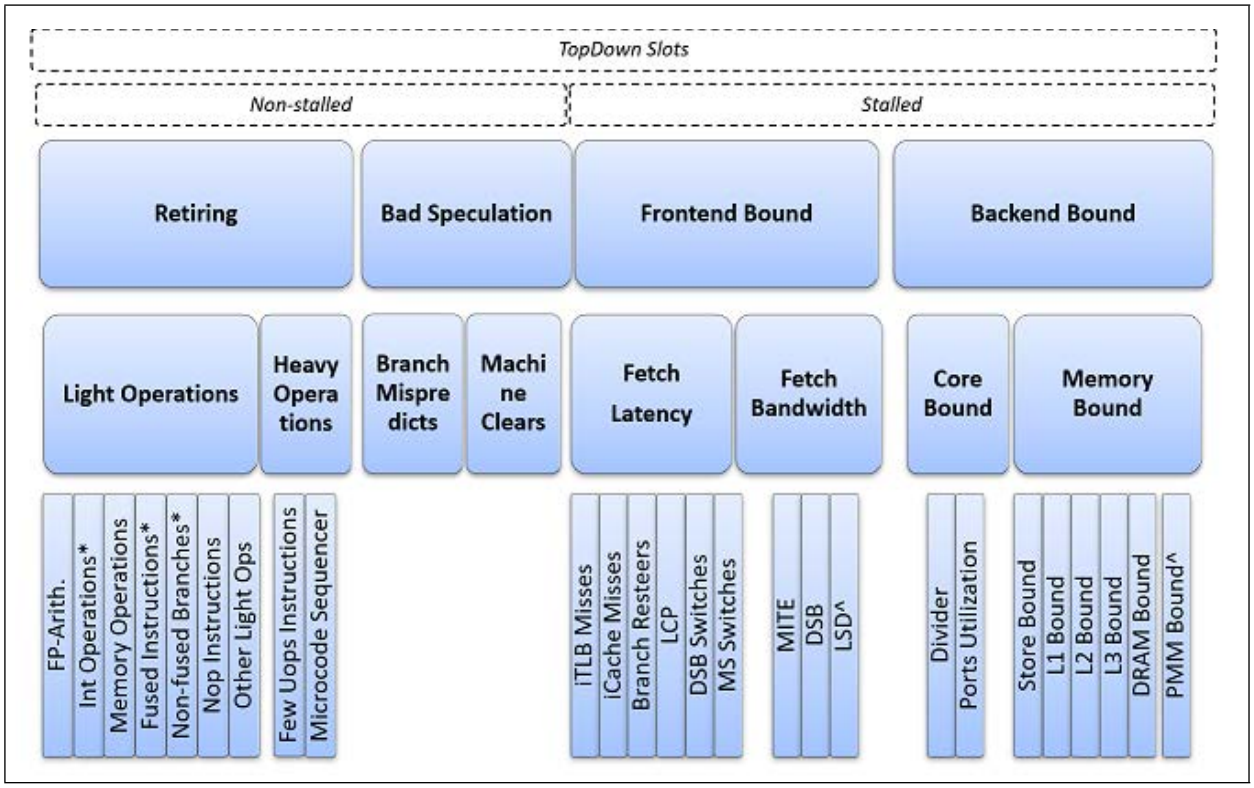


Figure 22-1. General TMA Hierarchy for Out-of-Order Microarchitectures

For example, if instruction fetch issues significantly hurt an application, TMA categorizes it as Frontend Bound at the tree's top level. A user/tool can drill down and focus only on the Frontend sub-tree. The drill down is recursively performed until a tree-leaf is reached. A leaf can point to a specific stall of the workload or denote a subset of issues with a common micro-architectural symptom likely to limit the application's performance.

TMA was first developed[1] in conjunction with the performance monitoring capability of the Sandy Bridge microarchitecture. The methodology is refined with subsequent generations to support multiple microarchitecture generations and enhanced by subsequent PMU capabilities. Please refer to the TMA electronic file at https://download.01.org/perfmon/ for details on the complete hierarchy and its nodes, additional useful, informative metrics, metric descriptions, event ratios per generation, or specific events.

22.1.1 TOP-LEVEL

At the top-level, TMA classifies pipeline slots into four main categories:

- Frontend Bound
- Backend Bound
- Bad Speculation
- Retiring

The latter two denote non-stalled slots while the former two indicate stalls, as illustrated in Figure 22-1 above. Figure 22-2 depicts a simple decision tree to start the drill-down process. If some operation utilizes a slot, it will be classified as Retiring or Bad Speculation, depending on whether it eventually gets retired (committed).

1. A Top-Down Method for Performance Analysis and Counters Architecture, Ahmad Yasin. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014. http://bit.ly/tma-ispass14 .

— Unused slots are classified as Backend Bound if the back end portion of the pipeline is unable to accept more operations (a.k.a. back-end stall[1]), or

— Frontend Bound: indicating no operations (µops) delivered while there was no back-end stall.

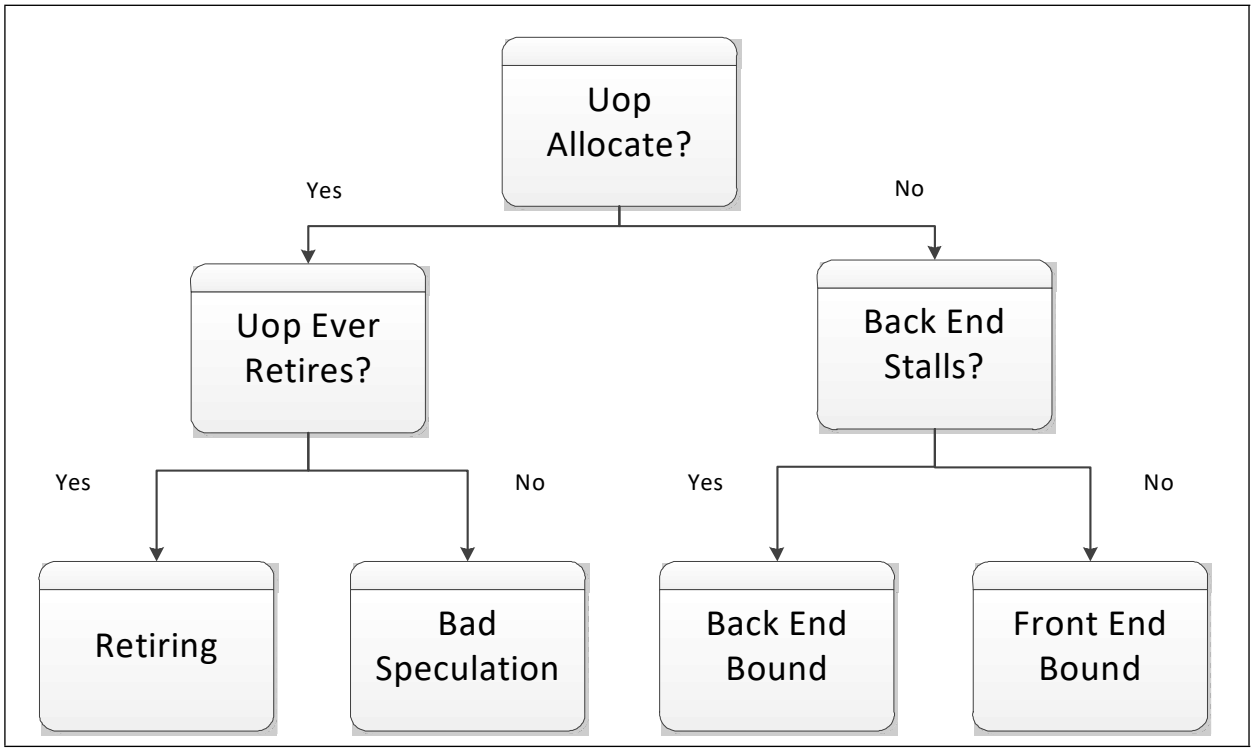


Figure 22-2. TMA's Top Level Drill Down Flowchart

The single entry point of division at a pipeline's issue stage (allocation stage) makes the four categories additive to the total possible slots. The classification at slots granularity (sub-cycle) makes the breakdown very accurate and robust for superscalar cores, which is necessary at the top level.

Retiring denotes slots utilized by "good operations." Ideally, you want to see all slots attributed here since it correlates with Instructions Per Cycle (IPC). Nevertheless, a high Retiring fraction does not necessarily mean there is no room for speedup.

Bad Speculation denotes slots wasted due to all aspects of incorrect speculations. It includes: (a) slots of operations that do not eventually retire and (b) slots where the issue pipeline was blocked due to recovery from earlier mis-speculations. Note there is a third portion covered by Branch_Resteers[1]. This category can be split per type of speculation. For example, Branch Mispredicts and Machine Clears cover control-flow and data mis-speculation, respectively.

Frontend Bound denotes when the pipeline's Frontend under-supplies the Backend. The Frontend is the pipeline portion responsible for delivering operations to be executed later by the Backend. This category is further classified into Fetch Latency (for example, ICache or ITLB misses) and Fetch Bandwidth (for instance, sub-optimal decoding).

Backend Bound denotes remaining stalled slots due to a lack of required Backend resources. It is split into Memory Bound, which reflects execution stalls due to the memory subsystem, and Core Bound, which demonstrates either pressure on the execution units (compute bound) or lack of Instructions-Level-Parallelism (ILP).

1. ibid.

The following sections provide more details on these categories and nodes in subsequent levels of the hierarchy.

22.1.2 FRONTEND BOUND

The Frontend denotes the pipeline portion where the branch predictor predicts the next address to fetch, streams of code bytes are fetched from ICache, parsed into instructions, and decoded into micro-ops that can be executed later by the back end. Frontend Bound denotes when the Frontend of the processor core under-supplies the Backend. There were fetch bubbles when the Backend was ready to accept µops (micro-ops).

Dealing with Frontend issues is tricky without TMA, as they occur at the beginning of the long and buffered pipeline. This often means transient issues will not dominate the actual performance, and you should investigate these issues only when Frontend Bound is flagged at the top level. In many cases, the front-end supply bandwidth can dominate the performance, especially when high IPC applies. This has led to the addition of dedicated units to hide the fetch pipeline latency and sustain required bandwidth, such as the Loop Stream Detector (LSD) and Decoded ICache (DSB).

TMA further distinguishes between latency and bandwidth Frontend stalls:

- An ICache miss is classified under **Fetch Latency**.
- Inefficiency in the instruction decoders is classified under **Fetch Bandwidth**.

Note that these metrics are defined in the top-down approach: **Fetch Latency** accounts for cases that lead to fetch starvation (the symptom of no µop delivery) regardless of what has caused that. Familiar i-cache and i-TLB misses fit here, but not only these. **Branch Resteers** accounts for fetch delays following pipeline flushes. Pipeline flushes can be caused by clear events such as branch misprediction or memory nukes. **Branch Resteers** are tightly coupled with Bad Speculation.

The methodology further classifies bandwidth issues per fetch unit, inserting µops to the Micro-Op-Queue (see Figure 2-6). Instruction decoders translate commonly-used x86 instructions into micro-ops that the rest of the machine understands; that would be one fetch unit. Some x86 instructions require sophisticated micro-op flows, like CPUID, relying on the MSROM to supply the long micro-op flows; that would be the 2nd fetch unit, and so on. Different fetch units may have different supply bandwidths from one generation to another. Figure 2-6 provides additional details for the Skylake microarchitecture.

All products do not support the LSD (^) node in Figure 22-1.

22.1.3 BACKEND BOUND

Backend Bound reflects slots where no micro-ops are being delivered at the issue pipeline, due to a lack of required resources for accepting them in the back end. Examples of performance issues in this category include data-cache misses or stalls due to the overloaded divider unit.

Backend Bound is split into **Memory Bound** and **Core Bound**. This is achieved by breaking down back-end stalls based on execution units' occupation at every cycle. To sustain a maximum IPC, it is necessary to keep execution units busy. For example, in a four-slot-wide machine, if three or fewer micro-ops are executed in a steady state of some code, this would prevent it from achieving an optimal IPC of 4. These sub-optimal cycles are called ExecutionStalls.

22.1.4 MEMORY BOUND

Memory Bound corresponds to execution stalls related to the cache and memory subsystems. These stalls usually manifest with execution units starved after a short while, like in the case of a load missing all caches. Many recent generations of Intel Core processors have three levels of cache hierarchy to hide external memory latency. The first level has a data cache (L1D). L2 is the second level shared instruction and data cache, which is private to each core. L3 is shared among all the processor cores within a physical package.

The out-of-order scheduler can dispatch micro-ops into multiple execution units for execution. While these micro-ops were executing in-flight, some of the memory access latency exposure for data can be hidden by keeping the execution units busy with useful micro-ops that do not depend on pending memory accesses. Thus for common cases, the real penalty for memory access is when the scheduler has nothing ready to feed the execution units. It is likely that further micro-ops are either waiting for the pending memory access or depend on other unready micro-ops.

ExecutionStalls span several sub-categories, each associated with a particular cache level and depending on the demanded data satisfied by the respective cache level. In some situations, an ExecutionStall can experience significant delay, greater than the nominal latency of the corresponding cache level, while no demand-load is missing that cache level.

For example, the L1D cache often has short latency which is comparable to ALU stalls (or waiting for completion of some commonly-used execution units like floating-point adds/multiplies or integer multiplies). Yet in certain scenarios, like a load blocked from forward data from an earlier store to an overlapping address, this load might suffer a high effective latency while eventually being satisfied by L1D. In such a scenario, the in-flight load will last a long time without missing L1D. Hence, it gets tagged under L1 Bound. Load blocks due to 4K Aliasing is another scenario with the same symptom.

ExecutionStalls related to store operations are also treated in the **Store Bound** category. Store operations are buffered and executed post-retirement due to memory ordering requirements. Typically, store operations have little impact on performance, but they cannot be neglected entirely. TMA defines Stores Bound as a fraction of cycles with low execution ports utilization and a high number of stores consuming resources needed to buff the stores.

Data TLB misses are categorized under various Memory Bound sub-nodes. For example, if a TLB translation is satisfied by L1D, it is tagged under **L1 Bound**.

A simple heuristic is used to distinguish **MEM Bandwidth** and **MEM Latency** under **DRAM Bound**. The heuristic uses the occupancy of requests pending on data return from the memory controller. Whenever the occupancy exceeds a high threshold, say 70% of the max number of requests, the memory controller can serve simultaneously; TMA flags this as potentially limited by memory bandwidth. The remainder fraction will be attributed to memory latency.

22.1.5 CORE BOUND

Core Bound corresponds to pressure on the execution units or lack of Instructions-Level-Parallelism (ILP) in your program. Core bound stalls can either manifest with short execution starvation periods, or with sub-optimal execution port utilization, which makes it more challenging to identify. For example, a long latency divide operation might serialize execution. In contrast, pressure on an execution port that serves specific varieties of micro-ops might manifest as a small number of ports utilized in a cycle.

Core Bound issues can often be mitigated with better code generation. For example, a sequence of dependent arithmetic operations would be classified as Core Bound. A compiler may relieve this stall with better instruction scheduling. Vectorization can mitigate Core Bound issues as well.

22.1.6 BAD SPECULATION

Bad Speculation reflects slots wasted due to incorrect speculations. These include two portions:

- Slots used to issue micro-ops that do not eventually retire.
- Slots in which the issue pipeline was blocked due to recovery from earlier mis-speculations.

For example, this category accounts for micro-ops issued in the shadow of a mispredicted branch. Note the third portion of a misprediction penalty deals with how quick is the fetch from the correct target. This is accounted for in **Branch Resteers** as it may overlap with other front-end stalls.

Having a Bad Speculation category at the Top Level is a crucial principle in TMA. It determines the fraction of the workload under analysis that is affected by incorrect execution paths, which in turn dictates the accuracy of observations listed in other categories. Furthermore, this permits nodes at lower levels to use of some of the many traditional performance counter events, despite most of those counter events counting speculatively. Hence, it would be best if you treated a high value in Bad Speculation as a "red flag" that needs to be investigated before looking at other categories. In other words, minimizing Bad Speculation improves the processor resource utilization and increases confidence in metrics reported throughout the hierarchy.

TMA further classifies Bad Speculation into **Branch Mispredicts** and **Machine Clears**, with similar symptoms where the pipeline is flushed. Branch misprediction applies when the BPU incorrectly predicts the branch direction and target. Memory Order Machine Clears (for example, due to memory disambiguation) are a subset of Machine Clears. The next steps to the analysis of these issues can be completely different—the first deals with making the program control flow friendlier to the branch predictor. The latter often points to unexpected situations such as memory ordering machine clears or self-modifying code.

22.1.7 RETIRING

This category reflects slots utilized by "good micro-ops" – issued micro-ops that get retired expeditiously without performance bottlenecks. Ideally, all slots attributed to the Retiring category should be seen; that is, Retiring 100% of slots corresponds to hitting the maximal micro-ops retired per cycle of the given microarchitecture. For example, assuming one instruction is decoded into one micro-op, Retiring of 50% in one slot means an IPC of 2 was achieved in a four-wide machine. In other words, maximizing the Retiring category increases the IPC of your program.

Nevertheless, a high Retiring value does not necessarily mean there is no room for more performance. Heavy Operations, like Floating Point (FP) assists, typically hurt performance and can be avoided. They are isolated under **Microcode Sequencer** to bring it to your attention.

A high Retiring value for non-vectorized code may be an excellent hint to vectorize the code. Doing so lets more operations to be completed by single instruction/micro-op, hence improving performance. TMA further breaks down the Retiring->Light Operations category into **FP Arith**, with **FP Scalar** and **FP Vector** operations distinction in level 4 (omitted in Figure B-1). For more details see the Matrix-Multiply use-case of the paper[2].

22.1.8 GOLDEN COVE MICROARCHITECTURE

All nodes with asterisks (*) in Figures 22-1 are introduced by the Golden Cove microarchitecture.

22.1.9 ICE LAKE MICROARCHITECTURE

The Ice Lake Microarchitecture supports a single "Branch Instructions" node replacing "Fused Instructions" and "Non-fused Branches" in Figure B-1.

22.1.10 OPTANE PERSISTENT MEMORY

The App Direct Mode the Intel® Optane™ DC Persistent Memory Modules introduced by the Cascade Lake server products are supported through the PMM_Bound node (^) in Figure B-1.

22.1.11 SKYLAKE MICROARCHITECTURE

The performance monitoring capabilities in the Skylake microarchitecture is significantly enhanced over prior generations. TMA benefits directly from the enhancement in the breadth of available counter events and in

Processor-Event-Based Sampling (PEBS) capabilities. TMAMTMA hierarchy supported by the Skylake microarchitecture's support for TMA, where the boxes in green indicates Precise events are available.

The Intel Vtune Performance Analyzer allows users to apply TMA on many Intel microarchitectures. The reader may wish to consult the white paper available on the Intel VTune Profiler Performance Analysis Cookbook for additional details.

22.1.11.1 TMA Use Case 1

Section 15.5.1 describes techniques for optimizing floating-point calculations involving latency and throughput considerations of FP MUL, FP ADD and FMA instructions. There are no explicit performance counter events that can directly detect exposures of latency issues of FP_ADD and FP_MUL instructions.

TMA may be used to figure out when this performance issue is likely to be a performance limiter.

If the primary bottleneck is Backend_Bound->Core_Bound->Ports_Utilization and there is a significant measure in the GFLOPs metric, the user code may be hitting this issue. The user may consider optimizations listed in Section 15.5.1.

22.1.11.2 TMA Use Case 2

Section 15.3.1 describes possible performance issues of executing Intel SSE code while the upper YMM state is dirty in Skylake Microarchitecture. To detect the performance issue associated with partial register dependence and associated blend cost on Intel SSE code execution, TMA can be used to monitor the rate of mixture of SSE operation and Intel AVX operation on performance-critical Intel SSE code whose source code did not directly execute Intel AVX instructions.

If the primary bottleneck is Backend_Bound->Core_Bound, and there is a significant measure in the Mixing_Vectors metric, it is possible that the presence of Vector operation with mis-matched vector width was due to the extra blend operation on the upper YMM registers.

The Mixing_Vectors metric requires the UOPS_ISSUED.VECTOR_WIDTH_MISMATCH event that is available in the Skylake Microarchitecture. This event count Blend µops was inserted at the issue stage to preserve upper bits of vector registers. Additionally, the metric uses the UOPS_ISSUED.ANY, which is common in recent Intel microarchitectures, as the denominator. event counts the total number of µops at the issue stage.

The Mixing_Vectors metric gives the percentage of injected blend µops out of all µops issued. Usually, a Mixing_Vectors over 5% is worth investigating.

Mixing_Vectors[%] = 100 * UOPS_ISSUED.VECTOR_WIDTH_MISMATCH / UOPS_ISSUED.ANY

Note that the actual penalty may vary as it stems from the additional data-dependency on the destination register the injected blend operations add.

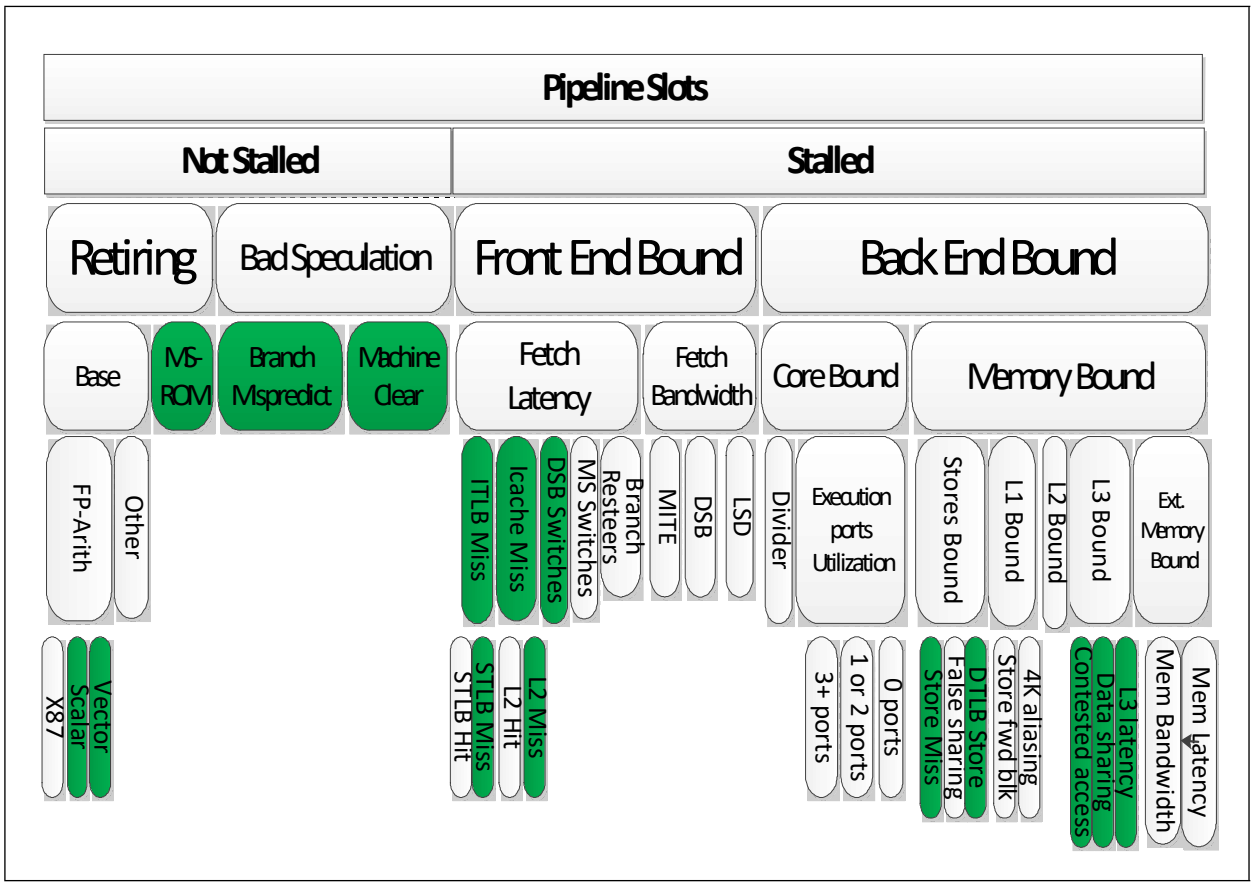


Figure 22-3. TMA Hierarchy and Precise Events in the Skylake Microarchitecture

22.2 PERFORMANCE MONITORING AND MICROARCHITECTURE

This section provides information on performance monitoring hardware and terminology related to the Silvermont, Airmont and Goldmont microarchitectures. The features described here may be specific to individual microarchitecture, as indicated in Table 22-1.

Table 22-1. Performance Monitoring Taxonomy

Name	Description	Applicable Microarchitectures
L2Q, XQ	When a memory reference misses the L1 data cache, the request goes to the L2 Queue (L2Q). If the request also misses the L2 cache, it is sent to the XQ, where it waits for an opportunity to be issued to memory across the Intra-Die Interface (IDI) link. Note that since the L2 is shared between a pair of processor cores, a single L2Q is shared between those two cores. Similarly, a single XQ for a pair of processor cores is situated between the L2Q and the IDI link. The XQ will fill up when the response rate from the IDI link is lower, at which new requests arrive at the XQ. The event L2_reject_XQ indicates that a request cannot move from the L2 Queue to the XQ because the XQ is full, signaling that the memory system is oversubscribed.	Silvermont, Airmont, Goldmont
core_reject	The core_reject event indicates that a request from the core cannot be accepted at the L2Q. However, there are several additional reasons why a request might be rejected from the L2Q. Beyond rejecting a request because the L2Q is full, a request from one core can be rejected to maintain fairness to the other core. One core is not permitted to monopolize the shared connection to the L2Q/cache/XQ/IDI links, and might have its requests rejected even when room is available in the L2Q. In addition, if the request from the core is a dirty L1 cache eviction, the hardware must ensure that this eviction does not conflict with any pending request in the L2Q. (pending requests can include an external snoop). In a conflict event, the dirty eviction request might be rejected even when there is room in the L2Q. Thus, while the L2_reject_XQ event indicates that the request rate to memory from both cores exceeds the response rate of the memory, the core_reject event is more subtle. It can either indicate that the request rate to the L2Q exceeds the response rate from the XQ, or that the request rate to the L2Q exceeds the response rate from the L2. It can also either indicate that one core is attempting to request more than its fair share of response from the L2Q, or be an indicator of conflict between dirty evictions and other pending requests. In short, the L2_reject_XQ event indicates memory oversubscription. The core_reject event can indicate memory oversubscription, L2 oversubscription, rejection of a core's requests to insure fairness to the other core, or a conflict between dirty evictions and other pending requests.	Silvermont, Airmont, Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
CYCLES_DIV_BUSY.ANY	Divider Busy: The divide unit cannot accept a new divide µop when it is busy processing a previously dispatched divide µop. The "CYCLES_DIV_BUSY.ANY" event will count cycles that the divide unit is busy, irrespective of whether or not another divide µop is waiting to enter the divide unit (from the RS). The event will count cycles while a divide is in progress, even if the RS is empty.	Silvermont, Airmont, Goldmont
BACLEAR	Shortly after decoding an instruction and recognizing a branch/call/jump/ret instruction, a Branch Address Calculator Clear (BACLEAR) event can occur. Possible causes of a BACLEAR include predicting the wrong target of a direct branch or not predicting a branch at that instruction location. A BACLEAR causes the Frontend to restart fetching from a different location. While BACLEAR has similarities to a branch mispredict signaled from the execute part of the pipeline, it is not counted as a BR_MISP_RETIRED event or noted as a mispredict in the LBRs (where LBRs report mispredict). Branch mispredicts and BACLEARS are similar in that they restart the Frontend to begin instruction fetch at a new target location and flush some speculative work. However, a branch mispredict must flush partially completed instructions from both the Frontend and back end. Since a BACLEAR occurs right at decode time, it flushes instruction bytes and not yet fully decoded instructions. Recovery after a BACLEAR is less complicated and faster than recovery after a branch mispredict.	Silvermont, Airmont, Goldmont
Front End Bottleneck	The front end is responsible for fetching the instruction, decoding it into micro-ops (µops) and putting those µops into a micro-op queue to be consumed by the back end. The back end then takes these micro-ops and allocates the required resources. When all resources are ready, micro-ops are executed. A front end bottleneck occurs when the front end of the machine is not delivering µops to the back-end, and the back end is not stalled. Cycles where the back end is not ready to accept micro-ops from the front end should not be counted as front end bottlenecks even though such back end bottlenecks will cause allocation unit stalls, eventually forcing the front end to wait until the back end is ready to receive more µops.	Silvermont, Airmont, Goldmont
NO_ALLOC_CYCLES	Frontend issues can be analyzed using various sub-events within this event class.	Silvermont, Airmont
UOPS_NOT_DELIVERED.ANY	The UOPS_NOT_DELIVERED.ANY event measures front end inefficiencies to identify if the machine is truly front end bound. Some examples of Frontend inefficiencies are: ICache misses, ITLB misses, and decoder restrictions that limit the Frontend bandwidth.	Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
ICache	Requests to Instruction Cache (ICache) are made in a fixed-size unit called a chunk. There are multiple chunks in a cache line, and multiple accesses might be made to a single cache line. In the Goldmont microarchitecture, the event strives to count on a cache line basis so that multiple fetches to a single cache line count as one ICACHE.ACCESS and either one HIT or one MISS. The event counts specifically when straight line code crosses the cache line boundary, or when a branch target is on a new line. This event is highly speculative, with bytes being fetched before being decoded, executed or retired. The speculation occurs in straight line code and in the presence of branches. Consequently, ICACHE statistics cannot be deduced by examining the number of retired instructions. In the Silvermont microarchitecture, ICACHE events (HIT, MISS) count at a different granularity.	Goldmont
ICache Access	An ICache fetch accesses an fixed-size aligned chunk. A request to fetch a specific chunk from the instruction cache might occur multiple times due to speculative execution. It may be possible that the same chunks requested multiple times while outstanding. However, an instruction fetch miss is only counted once and is not counted every cycle while outstanding. After an ICache miss fetches the line, another request to the same cache line is likely to be made and counted as a hit. The number “hits” plus “misses” does therefore not equal to the number of accesses. From a software perspective, the ICache miss count should be subtracted from the ICache hit count to get her number of true ICache hits.	Silvermont, Airmont, Goldmont
Last Level Cache References, Misses	On processors that do not have L3, L2 is the last level cache. The architectural performance event to count LLC references and misses are also known as L2_REQUESTS.ANY and L2_REQUESTS.MISS.	Silvermont, Airmont, Goldmont
Machine Clear	Many conditions might cause a machine clear, including the receipt of an interrupt, or a trap or a fault. All such conditions, including but not limited to Memory Ordering (MO), Self or Cross Modifying Code (SMC) and Floating Point assist (FP) are captured in the MACHINE_CLEAR.ANY event. Additionally, some conditions can be specifically counted (i.e. SMC, MO, FP). However, the sum of SMC, MO and FP machine clears will not necessarily equal the number of ANY.	Silvermont, Airmont, Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
MACHINE_CLEAR.FP_ASSIST	The floating point execute unit can properly produce the correct output bits most of the time. On rare occasions it needs help. A machine clear is asserted against the instruction to provide that help. After the machine clear, the front end of the machine starts delivering instructions to determine which FP operation was asked for. The instructions will perform extra work to produce the correct FP result. For example, if the result was a floating point denormal, sometimes the hardware asks the help to produce the correctly rounded IEEE compliant result.	Silvermont, Airmont, Goldmont
MACHINE_CLEAR.SMC	Self Modifying Code (SMC) refers to a piece of code that wrote to the instruction stream ahead of where the machine will execute. In the Silvermont microarchitecture, the processor detects SMC in a 1K aligned region. A detected SMC condition causes a machine clear assist and will flush the pipeline. Writing to memory within 1K of where the processor is executing can trigger the SMC detection mechanism and cause a machine clear. Since the machine clear allows the store pipeline to drain, when a front end restart occurs, the correct instructions after the write will be executed.	Silvermont, Airmont, Goldmont
MACHINE_CLEAR.MO	Memory order machine clear happens when a snoop request occurs and the machine is uncertain if memory ordering will be preserved. For instance, consider two loads: one to address X followed by another to address Y in the program order. Both loads were issued; however, load to Y completes first. and all the dependent ops and data on and by this load continue together. Load to X waits for the data. Simultaneously, another processor writes to the same address Y and causes a snoop to address Y. This presents a problem. The load to Y received the old value, but X is not finished loading. The other processor saw the loads in a different order by not consuming the latest value from the store to address Y. Everything from the load must be undone to address Y so the post-write data may be seen. Note: Without other pending reads, load Y does not require undoing. The ordering problem is caused by the unfinished load to X.	Silvermont, Airmont, Goldmont
MACHINE_CLEAR.DISAMBIGUATION	Disambiguation machine clear is triggered due to a younger load passing an older store to the same address, but whose address wasn't known when the younger load executed speculatively.	Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
Page Walk	When a translation of linear address to physical address cannot be found in the Translation Look-aside Buffer (TLB), dedicated hardware must retrieve the physical address from the page table and other paging structures if needed. After the page walk, the translation is stored in the TLB for future use. Since paging structures are stored in memory, the page walk can require multiple memory accesses. These accesses are considered part of demand data even if the page walk is to translate an instruction reference. The number of cycles for a page walk is variable, depending on how many memory accesses are required and the cache locality of those memory accesses. The PAGE_WALKS event can be used to count page walk durations with EDGE triger bit cleared. Page walk duration divided by number of page walks is the average duration of page-walks. In the Goldmont microarchitecture, the number of page walks can be determined by using the events MEM_UOPS_RETIRED.DTLB_MISS and ITLB.MISS. In the Silvermont microarchitecture, the combined number of page walks for data and instruction can be counted with PAGE_WALKS.WALKS.	Silvermont, Airmont, Goldmont
RAT	The allocation pipeline moves µops from the Frontend to the back end. At the end of the allocated pipe, a µop must written into one of 6 reservation stations (the RS). Each RS holds µops to be sent to a specific execution (or memory) cluster. Each RS has a finite capacity and may accumulate µops when it cannot send a µop to its execution cluster. Typical reasons an RS may fill include, but are not limited to, the execution of long latency µops like divide, the inability to schedule µops due to dependencies, or too many outstanding memory references. When the RS becomes full, it cannot accept more µops, and it will stall the allocation pipeline. The RS_FULL_STALL.ANY event will be asserted on any cycle when the allocation is stalled for any RSs being full and not for other reasons. (i.e., the allocated pipeline might be stalled for some other reason, but if RS is not full, the RS_FULL_STALL.ANY will not count). The MEC sub-event allows discovery of whether the MEC RS being full prevents further allocation.	Silvermont, Airmont, Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
REHABQ	An internal queue holds memory reference micro-ops that cannot complete for one reason or another. The micro-ops remain in the REHABQ until they can be re-issued and completed. Examples of bottlenecks that cause micro-ops to go into REHABQ include, but are not limited to: cache line splits, blocked store forward and data not ready. Many other conditions that might cause a load or store to be sent to the REHABQ. For instance, if an older store has an unknown address, all subsequent stores must be sent to the REHABQ until that older store's address becomes known.	Silvermont, Airmont
LOAD_BLOCKS	Loads can be blocked for multiple reasons, including UTLB misses, blocked store forwards, 4-K aliases or other conditions. When a load needs data (in whole or part) that a previous store produced, a forward progress of the machine will face two scenarios. The first, wherein the machine waits until the previous store is complete (forwarding restricted, loads blocked). In the second, data can be forwarded to the load before the previous store is complete. The restricted situations are described next. When a load is checked against previous stores, not all of its address bits are compared to the store addresses. This can cause a load to be blocked because its address is similar (LD_BLOCKS.4K_ALIAS) to a pending store, even though technically the load does not need to be blocked). When conditions do not allow the load to receive data from the in-progress store, then the load is blocked until the pending store operation is complete. LD_BLOCKS.STORE_FORWARD counts times when a load was prohibited from receiving forwarded data from the store because of address mismatch (explained below). LD_BLOCKS.DATA_UNKOWN counts when a load is blocked from using a store forward because the store data was unavailable at the right time. A load block will not be counted as both LD_BLOCKS.DATA_UNKNOWN and LD_BLOCK.STORE_FORWARD. These are precise events and thus will not count speculative loads that do not retire.	Goldmont
UOPS_RETIRED	The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences of µops; for example, floating point transcendental instructions, assists, and rep string instructions. In some cases, micro-op sequences are fused, or whole instructions are fused, into one micro-op. A sub-event within UOPS_RETIRED is available for differentiating MSROM micro-ops on Goldmont. The available sub-events differ from other microarchitectures.	Silvermont, Airmont, Goldmont

Table 22-1. Performance Monitoring Taxonomy (Contd.)

Name	Description	Applicable Microarchitectures
HW_INTERRUPTS	These Events provide information regarding Hardware (Vectored, Fixed) interrupts. HW_INTERRUPTS.RECEIVED provides a count of the total number of Hardware Interrupts received by the processor. This event is a straightforward count of the number of interrupts the ROB recognizes. HW_INTERRUPTS.PENDING_AND_MASKED counts the number of core cycles that an interrupt is pending but cannot be delivered due to EFLAGS.IF being 0. It will not count interrupts that TPR or ISR mask. These events are not precise, but collecting non-precise PEBS records on these events can help identify issues causing an unresponsive system.	Goldmont
MEM_UOPS_RETIRED	These events count when a µop reads (loads) or writes (stores) data if that µop retired valid. Speculative loads and stores are not counted. The sub-events can indicate conditions that generally require extra cycles to complete the operation: specifically, if the address of memory µop misses in the Data Translation Lookaside Buffer (DTLB), the data requested spans a cache line (split), or the memory µop is a locked load: these are precise events, so the EventingRIP field in the PEBS record indicates the instruction which caused the event.	Silvermont, Airmont, Goldmont
MEM_LOAD_UOPS_RETIRED	These events count when an instruction produces a µop that reads (loads) data if that µop is retired valid. Speculative loads are not counted. These events report the various states of the memory hierarchy for the requested data, which helps determine the source of latency stalls in accessing data. These are precise events, so the EventingRIP field in the PEBS record indicates the instruction which caused the event.	Goldmont

22.3 INTEL® XEON® PROCESSOR 5500 SERIES

Intel® Xeon® processor 5500 series are based on the same microarchitecture as Intel® Core i7 processors; see Section . In addition, the Intel Xeon processor 5500 series support non-uniform memory access (NUMA) in platforms with two physical processors; see Figure 22-4. Figure 22-4 illustrates four-processor cores and an uncore sub-system in each physical processor. The uncore sub-system consists of L3, an integrated memory controller (IMC), and Intel QuickPath Interconnect (QPI) interfaces. The memory sub-system consists of three channels of DDR3 memory locally connected to each IMC. Access to physical memory connected to a non-local IMC is often described as a remote memory access.

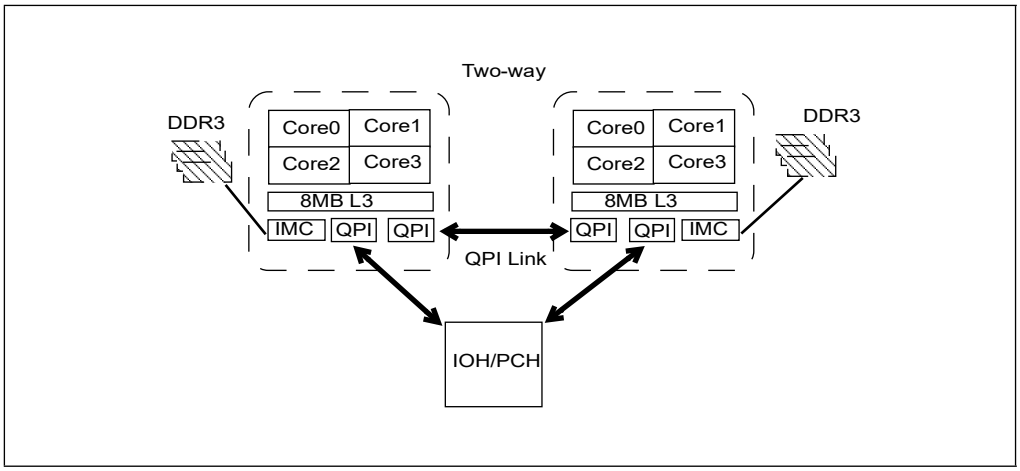


Figure 22-4. System Topology Supported by Intel® Xeon® Processor 5500 Series

The performance monitoring events on Intel Xeon processor 5500 series can be used to analyze the interaction between software (code and data) and microarchitectural units hierarchically:

- Per-core PMU: Each processor core provides four programmable counters and three fixed counters. The programmable per-core counters can be configured to investigate Frontend/micro-op flow issues and stalls inside a processor core. Additionally, a subset of per-core PMU events supports precise event-based sampling (PEBS). Load latency measurement facility is new in Intel Core i7 processor and Intel Xeon processor 5500.
- Uncore PMU: The uncore PMU provides eight programmable counters and one fixed counter. The programmable per-core counters can be configured to characterize L3 and Intel QPI operations and local and remote data memory accesses.

The number and variety of performance counters and the breadth of programmable performance events available in Intel Xeon processor 5500 offer software tuning engineers the ability to analyze performance issues and achieve higher performance. Using performance events to analyze performance issues can be grouped into the following subjects:

- Cycle Accounting and Uop Flow
- Stall Decomposition and Core Memory Access Events (non-PEBS)
- Precise Memory Access Events (PEBS)
- Precise Branch Events (PEBS, LBR)
- Core Memory Access Events (non-PEBS)
- Other Core Events (non-PEBS)
- Frontend Issues
- Uncore Events

22.4 PERFORMANCE ANALYSIS TECHNIQUES FOR INTEL® XEON® PROCESSOR 5500 SERIES

The techniques covered in this chapter focus on identifying an opportunity to remove/reduce performance bottlenecks that are measurable at runtime. Compile-time and source-code level techniques are covered in other chapters in this document. Individual sub-sections describe specific methods to identify tuning opportunities by examining various metrics that can be measured or derived directly from performance monitoring events.

22.4.1 CYCLE ACCOUNTING AND UOP FLOW ANALYSIS

The objectives, performance metrics and component events of the basic cycle accounting technique are summarized in Table 22-2.

Table 22-2. Cycle Accounting and Micro-ops Flow Recipe

	Summary
Objective	Identify code/basic block that had significant stalls
Method	Binary decomposition of cycles into “productive“ and “unproductive“ parts
PMU-Pipeline Focus	Micro-ops issued to execute
Event code/Umask	Event code B1H, Umask= 3FH for micro-op execution; Event code 3CH, Umak= 1, CMask=2 for counting total cycles
EvtSelc	Use CMask, Invert, Edge fields to count cycles and separate stalled vs. active cycles
Basic Equation	“Total Cycles“ = UOPS_EXECUTED.CORE_STALLS_CYCLES + UOPS_EXECUTED.CORE_ACTIVE_CYCLES
Metric	UOPS_EXECUTED.CORE_STALLS_CYCLES / UOPS_EXECUTED.CORE_STALLS_COUNT
Drill-down scope	Counting: Workload; Sampling: basic block
Variations	Port 0,1, 5 cycle counting for computational micro-ops execution.

Cycle accounting of executed micro-ops is an effective technique to identify stalled cycles for performance tuning. Within the microarchitecture pipeline, the meaning of micro-ops being “issued,” “dispatched,” “executed,” “retired” has a precise definition. This is illustrated in Figure 22-5.

Cycles are divided into those where micro-ops are dispatched to the execution units and those where no micro-ops are dispatched, which are considered execution stalls.

“Total cycles” of execution for the code under test can be directly measured with CPU_CLK_UNHALTED.THREAD (event code 3CH, Umask= 1) and setting CMask = 2 and INV=1 in IA32_PERFEVTSELCn.

The signals used to count the memory access µops executed (ports 2, 3 and 4) are the only core events that cannot be counted per-logical processor. Thus, Event code B1H with Umask=3FH only counts on a per-core basis, and the entire execution stall cycles can only be evaluated on a per-core basis. If HT is disabled, conducting a per-thread analysis of micro-op flow cycle accounting presents no difficulty.

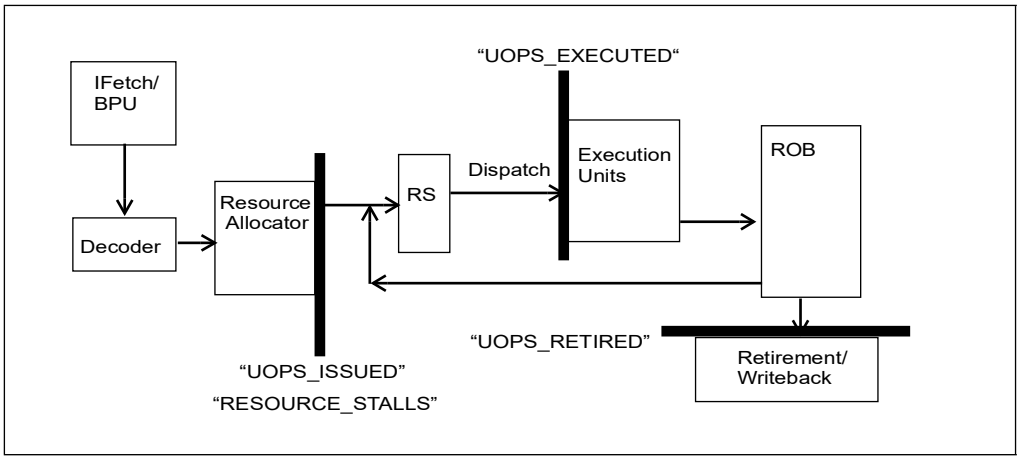


Figure 22-5. PMU Specific Event Logic Within the Pipeline

The PMU signals to count µops_executed in ports 0, 1, 5 can count on a per-thread basis even when HT is active. This provides an alternate cycle accounting technique when the workload under test interacts with HT.

The alternate metric is built from UOPS_EXECUTED.PORT015_STALL_CYCLES, using appropriate CMask, Inv, and Edge settings. Details of performance events are shown in Table 22-3.

Table 22-3. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow

Event Name	Umask	Event Code	CMask	Inv	Edge	All Thread
CPU_CLK_UNHALTED.TOTAL_CYCLES	0H	3CH	2	1	0	0
UOPS_EXECUTED.CORE_STALLS_CYCLES	3FH	B1H	1	1	0	1
UOPS_EXECUTED.CORE_STALLS_COUNT	3FH	B1H	1	1	!	1
UOPS_EXECUTED.CORE_ACTIVE_CYCLES	3FH	B1H	1	0	0	1
UOPS_EXECUTED.PORT015_STALLS_CYCLES	40H	B1H	1	1	0	0
UOPS_RETIRED.STALL_CYCLES	1H	C2H	1	1	0	0
UOPS_RETIRED.ACTIVE_CYCLES	1H	C2H	1	0	0	0

22.4.1.1 Cycle Drill Down and Branch Mispredictions

While executed micro-ops are considered productive from the perspective of execution units being subscribed, not all such micro-ops contribute to forward progress of the program. Branch mispredictions can introduce execution inefficiencies in OOO processor that are typically decomposed into three components:

- Wasted work associated with executing the µops of the incorrectly predicted path.
- Cycles lost when the pipeline is flushed of the incorrect µops.
- Cycles lost while waiting for the correct µops to arrive at the execution units.

In processors based on Nehalem microarchitecture, there are no execution stalls associated with clearing the pipeline of mispredicted µops (component 2). These µops are simply removed from the pipeline without stalling executions or dispatch. This typically lowers the penalty for mispredicted branches. Further, the penalty associated with instruction starvation (component 3) can be measured.

The wasted work within executed µops are those µops that will never be retired. This is part of the cost associated with mispredicted branches. It can be found through monitoring the flow of µops through the pipeline. The µop flow can be measured at 3 points in Figure 22-5, going into the RS with the event UOPS_ISSUED, going into the execution units with UOPS_EXECUTED and at retirement with UOPS_RETIRED. The differences of between the upstream measurements and at retirement measure the wasted work associated with these mispredicted µops.

As UOPS_EXECUTED must be measured per core, rather than per thread, the wasted work per core is evaluated as:

Wasted Work = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_All_Thread - UOPS_RETIRED.ANY_ALL_THREAD.

The ratio above can be converted to cycles by dividing the average issue rate of µops. The events above were designed to be used in this manner without corrections for micro fusion or macro fusion.

A “per thread” measurement can be made from the difference between the µops issued and µops retired as the latter two of the above events can be counted per thread. It over counts slightly, by the mispredicted µops that are eliminated in the RS before they can waste cycles being executed, but this is usually a small correction:

Wasted Work/thread = (UOPS_ISSUED.ANY + UOPS_ISSUED.FUSED) - UOPS_RETIRED.ANY.

Table 22-4. Cycle Accounting of Wasted Work Due to Misprediction

	Summary
Objective	Evaluate µops that executed but not retired due to misprediction
Method	Examine µop flow differences between execution and retirement
PMU-Pipeline Focus	Micro-ops execute and retirement
Event code/Umask	Event code B1H, Umask= 3FH for micro-op execution; Event code C2H, Umask= 1, AllThread=1 for per-core counting
EvtSelc	Zero CMask, Invert, Edge fields to count µops
Basic Equation	“Wasted work“ = UOPS_EXECUTED.PORT234_CORE + UOPS_EXECUTED.PORT015_ALL_THREAD - UOPS_RETIRED.ANY_ALL_THREAD
Drill-down scope	Counting: Branch misprediction cost
Variations	Divide by average µop issue rate for cycle accounting. Set AllThread=0 to estimate per-thread cost.

The third component of the misprediction penalty, instruction starvation, occurs when the instructions associated with the correct path are far away from the core and execution is stalled due to lack of µops in the RAT. Because the two primary cause of µops not being issued are

- Frontend starvation.
- Resources are unavailable in the back end.

So the output of the resource allocation can be measured as follows:

- Count the total number of cycles where no µops were issued to the OOO engine.
- Count the cycles where resources (RS, ROB entries, load buffer, store buffer, etc.) are unavailable for allocation.

If HT is not active, instruction starvation is simply the difference:

Instruction Starvation = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY.

When HT is enabled, the µop delivery to the RS alternates between the two threads. In an ideal case the above condition would then over count, as 50% of the issuing stall cycles may be delivering µops for the other thread. The expression can be modified by subtracting the cycles that the other thread is having µops issued.

Instruction Starvation (per thread) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD.

The per-thread expression above will over count somewhat because the resource_stall condition could exist on “this” thread while the other thread in the same core was issuing µops. An alternative might be:

CPU_CLK_UNHALTED.THREAD - UOPS_ISSUED.CORE_CYCLES_ACTIVE-RESOURCE_STALLS.ANY.

The above technique is summarized in Table 22-5.

Table 22-5. Cycle Accounting of Instruction Starvation

	Summary
Objective	Evaluate cycles that µops issuing is starved after misprediction
Method	Examine cycle differences between µops issuing and resource allocation
PMU-Pipeline Focus	Micro-ops issue and resource allocation
Event code/Umask	Event code 0EH, Umak= 1, for µops issued. Event code A2H, Umask=1, for Resource allocation stall cycles
EvtSelc	Set CMask=1, Inv=1, fields to count µops issue stall cycles. Set CMask=1, Inv=0, fields to count µops issue active cycles. Use AllThread = 0 and AllThread=1 on two counter to evaluate contribution from the other thread for UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD
Basic Equation	“Instruction Starvation“ (HT off) = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY;
Drill-down scope	Counting: Branch misprediction cost
Variations	Evaluate per-thread contribution with Instruction Starvation = UOPS_ISSUED.STALL_CYCLES - RESOURCE_STALLS.ANY - UOPS_ISSUED.ACTIVE_CYCLES_OTHER_THREAD

Details of performance events are shown in Table 22-6.

Table 22-6. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow

Event Name	Umask	Event Code	CMask	Inv	Edge	All Thread
UOPS_EXECUTED.PORT234_CORE	80H	B1H	0	0	0	1
UOPS_EXECUTED.PORT015_ALL_THREAD	40H	B1H	0	0	0	1
UOPS_RETIRED.ANY_ALL_THREAD	1H	C2H	0	0	0	1
RESOURCE_STALLS.ANY	1H	A2H	0	0	0	0
UOPS_ISSUED.ANY	1H	0EH	0	0	0	0
UOPS_ISSUED.STALL_CYCLES	1H	0EH	1	1	0	0
UOPS_ISSUED.ACTIVE_CYCLES	1H	0EH	1	0	0	0
UOPS_ISSUED.CORE_CYCLES_ACTIVE	1H	0EH	1	0	0	1

22.4.1.2 Basic Block Drill Down

The event INST_RETIRED.ANY (instructions retired) is commonly used to evaluate a cycles/instruction ratio (CPI). Another important usage is to determine the performance-critical basic blocks by evaluating basic block execution counts.

In a sampling tool (such as VTune Analyzer), the samples tend to cluster around certain IP values. This is true when using INST_RETIRED.ANY or cycle counting events. Disassembly listing based on the hot samples may associate some instructions with high sample counts and adjacent instructions with no samples.

Because all instructions within a basic block are retired exactly the same number of times by the very definition of a basic block. Drilling down the hot basic blocks will be more accurate by averaging the sample counts over the instructions of the basic block.

> Basic Block Execution Count = Sum (Sample counts of instructions within basic block) * Sample_after_value / (number of instructions in basic block)

Inspection of disassembly listing to identify basic blocks associated with loop structure being a hot loop or not can be done systematically by adapting the technique above to evaluate the trip count of each loop construct. For a simple loop with no conditional branches, the trip count ends up being the ratio of the basic block execution count of the loop block to the basic block execution count of the block immediately before and/or after the loop block. Judicious use of averaging over multiple blocks can be used to improve the accuracy.

This will allow the user to identify loops with high trip counts to focus on tuning efforts. This technique can be implemented using fixed counters.

Chains of dependent long-latency instructions (fmul, fadd, imul, etc) can result in the dispatch being stalled while the outputs of the long latency instructions become available. In general there are no events that assist in counting such stalls with the exception of instructions using the divide/sqrt execution unit. In such cases, the event ARITH can be used to count both the occurrences of these instructions and the duration in cycles that they kept their execution units occupied. The event ARITH.CYCLES_DIV_BUSY counts the cycles that either the divide/sqrt execution unit was occupied.

22.4.2 STALL CYCLE DECOMPOSITION AND CORE MEMORY ACCESSES

The decomposition of the stall cycles is accomplished through a standard approximation. It is assumed that the penalties occur sequentially for each performance impacting event. Consequently, the total loss of cycles available for useful work is then the number of events, N_i, times the average penalty for each type of event, P_i

- Counted_Stall_Cycles = Sum (N_i* P_i)list

This only accounts for the performance impacting events that are or can be counted with a PMU event. Ultimately there will be several sources of stalls that cannot be counted, however their total contribution can be estimated:

- Unaccounted stall cycles = Stall_Cycles - Counted_Stall_Cycles = UOPS_EXECUTED.CORE_STALLS_CYCLES - Sum (N_i* P_i)_both_threads

The unaccounted component can become negative as the sequential penalty model is overly simple and usually over counts the contributions of the individual microarchitectural issues.

As noted in Section 22.4.1.1, UOPS_EXECUTED.CORE_STALL_CYCLES counts on a per core basis rather than on a per thread basis, the over counting can become severe. In such cases it may be preferable to use the port 0,1,5 µop stalls, as that can be done on a per thread basis:

- Unaccounted stall cycles (per thread) = UOPS_EXECUTED.PORT015_THREADED_STALLS_CYCLES - Sum (N_i* P_i)

This unaccounted component is meant to represent the components that were either not counted due to lack of performance events or simply neglected during the data collection.

One can also choose to use the “retirement” point as the basis for stalls. The PEBS event, UOPS_RETIRED.STALL_CYCLES, has the advantage of being evaluated on a per thread basis and being having the HW capture the IP associated with the retiring µop. This means that the IP distribution will not be effected by STI/CLI deferral of interrupts in critical sections of OS kernels, thus producing a more accurate profile of OS activity.

22.4.2.1 Measuring Costs of Microarchitectural Conditions

Decomposition of stalled cycles in this manner should start by first focusing on conditions that carry large performance penalty, for example, events with penalties of greater than 10 cycles. Short penalty events (P < 5 cycles) can frequently be hidden by the combined actions of the OOO execution and the compiler. The OOO engine manages both types of situations in the instruction stream and strive to keep the execution units busy during stalls of either type due to instruction dependencies. Usually, the large penalty operations are dominated by memory access and the very long latency instructions for divide and sqrt.

The largest penalty events are associated with load operations that require a cacheline which is not in L1 or L2 of the cache hierarchy. Occurrences must be counted, and the penalty to be assigned must be known.

The standard approach to measuring latency is to measure the average number of cycles a request is in a queue:

- Latency = Sum (CYCLES_Queue_entries_outstanding) /Queue_inserts

where “queue_inserts“ refers to the total number of entries that caused the outstanding cycles in that queue. However, the penalty associated with each queue insert (i.e. cachemiss), is the latency divided by the average queue occupancy. This correction is needed to avoid over counting associated with overlapping penalties.

- Avg_Queue_Depth= Sum (CYCLES_Queue_entries_outstanding) / Cycles_Queue_not_empty

The the penalty (cost) of each occurrence is

- Penalty = Latency / Avg_Queue_Depth = Cycles_Queue_not_empty / Queue_inserts

An alternative way of thinking about this is to realize that the sum of all the penalties, for an event that occupies a queue for its duration, cannot exceed the time that the queue is not empty

- Cycles_Queue_not_empty = Events * <Penalty>

The standard techniques described above are simple conceptually. In practice, the large amount of memory references in the workload and wide range of varying state/location-specific latencies made standard sampling

techniques less practical. Using precise-event-based sampling (PEBS) is the preferred technique for processors based on Nehalem microarchitecture.

The profiling the penalty by sampling (to localize the measurement in IP) is likely to have accuracy difficulties. Since the latencies for L2 misses can vary from 40 to 400 cycles, collecting the number of required samples will tend to be invasive.

The use of the precise latency event, that will be discussed later, provides a more accurate and flexible measurement technique when sampling is used. As each sample records both a load to use latency and a data source, the average latency per data source can be evaluated. Further as the PEBS hardware supports buffering the events without generating a PMI until the buffer is full, it is possible to make such an evaluation efficient without perturbing the workload intrusively.

A number of performance events in core PMU can be used to measure the costs of memory accesses that originated in the core and experienced delays due to various conditions, locality, or traffic due to cache coherence requirements. The latency of memory accesses vary, depending on locality of L3, DRAM attached to the local memory controller or remote controller, and cache coherency factors. Some examples of the approximate latency values are shown in Table 22-7.

Table 22-7. Approximate Latency of L2 Misses of Intel Xeon Processor 5500

Data Source	Latency
L3 hit, Line exclusive	~ 42 cycles
L3 Hit, Line shared	~ 63 cycles
L3 Hit, modified in another core	~ 73 cycles
Remote L3	100 - 150 cycles
Local DRAM	~ 50 ns
Remote DRAM	~ 90 ns

22.4.3 CORE PMU PRECISE EVENTS

The Precise Event Based Sampling (PEBS) mechanism enables the PMU to capture the architectural state and IP at the completion of the instruction that caused the event. This provides two significant benefit for profiling and tuning:

- The location of the eventing condition in the instruction space can be accurate profiled,
- Instruction arguments can be reconstructed in a post processing phase, using captured PEBS records of the register states.

The PEBS capability has been greatly expanded in processors based on Nehalem microarchitecture, covering a large number of and more types of precise events.

The mechanism works by using the counter overflow to arm the PEBS data acquisition. Then on the next event, the data is captured and the interrupt is raised.

The captured IP value is sometimes referred to as IP +1, because at the completion of the instruction, the IP value is that of the next instruction.

By their very nature precise events must be “at-retirement” events. For the purposes of this discussion the precise events are divided into Memory Access events, associated with the retirement of loads and stores, and Execution Events, associated with the retirement of all instructions or specific non memory instructions (branches, FP assists, SSE µops).

22.4.3.1 Precise Memory Access Events

There are two important common properties to all precise memory access events:

- The exact instruction can be identified because the hardware captures the IP of the offending instruction. Of course the captured IP is that of the following instruction but one simply moves the samples up one instruction. This works even when the recorded IP points to the first instruction of a basic block because n such a case the offending instruction has to be the last instruction of the previous basic block, as branch instructions never load or store data, instruction arguments can be reconstructed in a post processing phase, using captured PEBS records of the register states.
- The PEBS buffer contains the values of all 16 general registers, R1-R16, where R1 is also called RAX. When coupled with the disassembly the address of the load or store can be reconstructed and used for data access profiling. The Intel® Performance Tuning Utility does exactly this, providing a wide variety of powerful analysis techniques

Precise memory access events mainly focus on loads as those are the events typically responsible for the very long duration execution stalls. They are broken down by the data source, thereby indicating the typical latency and the data locality in the intrinsically NUMA configurations. These precise load events are the only L2, L3 and DRAM access events that only count loads. All others will also include the L1D and/or L2 hardware prefetch requests. Many will also include RFO requests, both due to stores and to the hardware prefetchers.

All four general counters can be programmed to collect data for precise events. The ability to reconstruct the virtual addresses of the load and store instructions allows an analysis of the cacheline and page usage efficiency. Even though cachelines and pages are defined by physical address the lower order bits are identical, so the virtual address can be used.

As the PEBS mechanism captures the values of the register at completion of the instruction, one should be aware that pointer-chasing type of load operation will not be captured because it is not possible to infer the load instruction from the dereferenced address.

The basic PEBS memory access events falls into the following categories:

- MEM_INST_RETIRED: This category counts instruction retired which contain a load operation, it is selected by event code 0BH.
- MEM_LOAD_RETIRED: This category counts retired load instructions that experienced specific condition selected by the Umask value, the event code is 0CBH.
- MEM_UNCORE_RETIRED: This category counts memory instructions retired and received data from the uncore sub-system, it is selected by event code 0FH.
- MEM_STORE_RETIRED: This category counts instruction retired which contain a store operation, it is selected by event code 0CH.
- ITLE_MISS_RETIRED: This counts instruction retired which missed the ITLB, it is selected by event code 0C8H

Umask values and associated name suffixes for the above PEBS memory events are listed *on the Intel PerfMon Events page*.

The precise events listed above allow load driven cache misses to be identified by data source. This does not identify the “home” location of the cachelines with respect to the NUMA configuration. The exceptions to this statement are the events

MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.NON_LOCAL_DRAM. These can be used in conjunction with instrumented malloc invocations to identify the NUMA “home” for the critical contiguous buffers used in an application.

The sum of all the MEM_LOAD_RETIRED events will equal the MEM_INST_RETIRED.LOADS count.

A count of L1D misses can be achieved with the use of all the MEM_LOAD_RETIRED

events, except MEM_LOAD_RETIRED.L1D_HIT. It is better to use all of the individual MEM_LOAD_RETIRED events to do this, rather than the difference of MEM_INST_RETIRED.LOADS-MEM_LOAD_RETIRED.L1D_HIT because while the total counts of precise events will be correct, and they will correctly identify instructions that caused the event in question, the distribution of the events may not be correct due to PEBS SHADOWING, discussed later in this section.

L1D_MISSES = MEM_LOAD_RETIRED.HIT_LFB + MEM_LOAD_RETIRED.L2_HIT + MEM_LOAD_RETIRED.L3_UNSHARED_HIT + MEM_LOAD_RETIRED.OTHER_CORE_HIT_HITM + MEM_LOAD_RETIRED.L3_MISS

The MEM_LOAD_RETIRED.L3_UNSHARED_HIT event merits some explanation. The inclusive L3 has a bit pattern to identify which core has a copy of the line. If the only bit set is for the requesting core (unshared hit) then the line can be returned from the L3 with no snooping of the other cores. If multiple bits are set, then the line is in a shared state and the copy in the L3 is current and can also be returned without snooping the other cores.

If the line is read for ownership (RFO) by another core, this will put the copy in the L3 into an exclusive state. If the line is then modified by that core and later evicted, the written back copy in the L3 will be in a modified state and snooping will not be required. MEM_LOAD_RETIRED.L3_UNSHARED_HIT counts all of these. The event should really have been called MEM_LOAD_RETIRED.L3_HIT_NO_SNOOP.

The event MEM_LOAD_RETIRED.L3_HIT_OTHER_CORE_HIT_HITM could have been named as MEM_LOAD_RETIRED.L3_HIT_SNOOP intuitively for similar reason.

When a modified line is retrieved from another socket it is also written back to memory. This causes remote HITM access to appear as coming from the home dram. The MEM_UNCORE_RETIRED.LOCAL_DRAM and MEM_UNCORE_RETIRED.REMOTE_DRAM evens thus also count the L3 misses satisfied by modified lines in the caches of the remote socket.

There is a difference in the behavior of MEM_LOAD_RETIRED.DTLB_MISSES with respect to that on Intel® Core™2 processors. Previously the event only counted the first miss to the page, as do the imprecise events. The event now counts all loads that result in a miss, thus it includes the secondary misses as well.

22.4.3.2 Load Latency Event

Intel processors based on Nehalem microarchitecture provide support for “load-latency event”, MEM_INST_RETIRED with event code 0BH and Umask value of 10H (LATENCY_ABOVE_THRESHOLD). This event samples loads, recording the number of cycles between the execution of the instruction and actual deliver of the data. If the measured latency is larger than the minimum latency programmed into MSR 0x3f6, bits 15:0, then the counter is incremented.

Counter overflow arms the PEBS mechanism and on the next event satisfying the latency threshold, the PMU writes the measured latency, the virtual or linear address, and the data source into a PEBS record format in the PEBS buffer. Because the virtual address is captured into a known location, the sampling driver could also execute a virtual to physical translation and capture the physical address. The physical address identifies the NUMA home location and in principle allows an analysis of the details of the cache occupancies.

Further, as the address is captured before retirement even the pointer chasing encoding “MOV RAX, [RAX+const]” have their addresses captured. Because the MSR_PEBS_LD_LAT_THRESHOLD MSR is required to specify the latency threshold value, only one minimum latency value can be sampled on a core during a given period. To enable this, the Intel performance tools restrict the programming of this event to counter 4 to simplify the scheduling. Table 22-8 lists a few examples of event programming configurations used by the Intel® PTU and Vtune™ Performance Analyzer for the load latency events. Different threshold values for the minimum latencies are specified in MSR_PEBS_LD_LAT_THRESHOLD (address 0x3f6).

Table 22-8. Load Latency Event Programming

Load Latency Precise Events	MSR 0x3F6	Umask	Event Code
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_4	4	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8	8	10H	0BH

Table 22-8. Load Latency Event Programming (Contd.)

Load Latency Precise Events	MSR 0x3F6	Umask	Event Code
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_10	16	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_20	32	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_40	64	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_80	128	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_100	256	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_200	512	10H	0BH
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_8000	32768	10H	0BH

One of the three fields written to each PEBS record by the PEBS assist mechanism of the load latency event, encodes the data source locality information.

Table 22-9. Data Source Encoding for Load Latency PEBS Record

Encoding	Description
0x0	Unknown L3 cache miss.
0x1	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
0x2	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway. The data is not yet in the data cache, but is located in a fill buffer that will soon be committed to cache.
0x3	This data request was satisfied by the L2.
0x4	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
0x5	L3 HIT (other core hit snoop). Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (Clean).
0x6	L3 HIT (other core HITM). Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
0x7	Reserved
0x8	L3 MISS (remote cache forwarding). Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
0x9	Reserved.
0xA	L3 MISS (local DRMA go to S). Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0xB	L3 MISS (remote DRMA go to S). Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0xC	L3 MISS (local DRMA go to E). Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).

Table 22-9. Data Source Encoding for Load Latency PEBS Record (Contd.)

Encoding	Description
0xD	L3 MISS (remote DRMA go to E). Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0xE	I/O, Request of input/output operation.
0xF	The request was to uncacheable memory.

The latency event is the recommended method to measure the penalties for a cycle accounting decomposition. Each time a PMI is raised by this PEBS event a load to use latency and a data source for the cacheline is recorded in the PEBS buffer. The data source for the cacheline can be deduced from the low order 4 bits of the data source field and the table shown above. Thus an average latency for each of the 16 sources can be evaluated from the collected data. As only one minimum latency at a time can be collected it may be awkward to evaluate the latency for an MLC hit and a remote socket dram. A minimum latency of 32 cycles should give a reasonable distribution for all the off-core sources however. The Intel® PTU version 3.2 performance tool can display the latency distribution in the data profiling mode and allows sophisticated event filtering capabilities for this event.

22.4.3.3 Precise Execution Events

PEBS capability in core PMU goes beyond load and store instructions. Branches, near calls and conditional branches can all be counted with precise events, for both retired and mispredicted (and retired) branches of the type selected. For these events, the PEBS buffer will contain the target of the branch. If the Last Branch Record (LBR) is also captured then the location of the branch instruction can also be determined.

When the branch is taken the IP value in the PEBS buffer will also appear as the last target in the LBR. If the branch was not taken (conditional branches only) then it won't and the branch that was not taken and retired is the instruction before the IP in the PEBS buffer.

In the case of near calls retired, this means that Event Based Sampling (EBS) can be used to collect accurate function call counts. As this is the primary measurement for driving the decision to inline a function, this is an important improvement. to measure call counts, you must sample on calls. Any other trigger introduces a bias that cannot be guaranteed to be corrected properly.

The precise branch events can be found under event code C4H *at: https://perfmon-events.intel.com/*.

There is one source of sampling artifact associated with precise events. It is due to the time delay between the PMU counter overflow and the arming of the PEBS hardware. During this period events cannot be detected due to the timing shadow. To illustrate the effect, consider a function call chain where a long duration function, "foo", which calls a chain of 3 very short duration functions, "foo1" calling "foo2" which calls "foo3", followed by a long duration function "foo4". If the durations of foo1, foo2 and foo3 are less than the shadow period the distribution of PEBS sampled calls will be severely distorted. For example:

- If the overflow occurs on the call to foo, the PEBS mechanism is armed by the time the call to foo1 is executed and samples will be taken showing the call to foo1 from foo.
- If the overflow occurs due to the call to foo1, foo2 or foo3 however, the PEBS mechanism will not be armed until execution is in the body of foo4. Thus the calls to foo2, foo3 and foo4 cannot appear as PEBS sampled calls.

Shadowing can effect the distribution of all PEBS events. It will also effect the distribution of basic block execution counts identified by using the combination of a branch retired event (PEBS or not) and the last entry in the LBR. If there were no delay between the PMU counter overflow and the LBR freeze, the last LBR entry could be used to sample taken retired branches and from that the basic block execution counts. All the instructions between the last taken branch and the previous target are executed once.

Such a sampling could be used to generate a "software" instruction retired event with uniform sampling, which in turn can be used to identify basic block execution counts. Unfortunately the shadowing causes the branches at the

end of short basic blocks to not be the last entry in the LBR, distorting the measurement. Since all the instructions in a basic block are by definition executed the same number of times.

The shadowing effect on call counts and basic block execution counts can be alleviated to a large degree by averaging over the entries in the LBR. This will be discussed in the section on LBRs.

Typically, branches account for more than 10% of all instructions in a workload, loop optimization must focus on those loops with high tripcounts. For counted loops, it is very common for the induction variable to be compared to the tripcount in the termination condition evaluation. This is particularly true if the induction variable is used within the body of the loop, even in the face of heavy optimization. Thus a loop sequence of unrolled operation by eight times may resemble:

```
add     rcx, 8
cmp     rcx, rax
jnge    triad+0x27
```

In this case the two registers, rax and rcx are the tripcount and induction variable. If the PEBS buffer is captured for the conditional branches retired event, the average values of the two registers in the compare can be evaluated. The one with the larger average will be the tripcount. Thus the average, RMS, min and max can be evaluated and even a distribution of the recorded values.

22.4.3.4 Last Branch Record (LBR)

The LBR captures the source and target of each retired taken branch. Processors based on Nehalem microarchitecture can track 16 pairs of source/target addresses in a rotating buffer. Filtering of the branch instructions by types and privilege levels are permitted using a dedicated facility, MSR_LBR_SELECT. This means that the LBR mechanism can be programmed to capture branches occurring at ring 0 or ring 3 or both (default) privilege levels. Further the types of taken branches that are recorded can also be filtered.

The list of filtering options that can be specified using MSR_LBR_SELECT is described in Chapter 18, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

The default is to capture all branches at all privilege levels (all bits zero). Another reasonable programming would set all bits to 1 except bit 1 (capture ring 3) and bit 3 (capture near calls) and bits 6 and 7. This would leave only ring 3 calls and unconditional jumps in the LBR. Such a programming would result in the LBR having the last 16 taken calls and unconditional jumps retired and their targets in the buffer.

A PMU sampling driver could then capture this restricted “call chain” with any event, thereby providing a “call tree” context. The inclusion of the unconditional jumps will unfortunately cause problems, particularly when there are if-else structures within loops.

In the case of frequent function calls at all levels, the inclusion of returns could be added to clarify the context. However this would reduce the call chain depth that could be captured. A fairly obvious usage would be to trigger the sampling on extremely long latency loads, to enrich the sample with accesses to heavily contended locked variables, and then capture the call chain to identify the context of the lock usage.

Call Counts and Function Arguments

If the LBRs are captured for PMIs triggered by the BR_INST_RETIRED.NEAR_CALL event, then the call count per calling function can be determined by simply using the last entry in LBR. As the PEBS IP will equal the last target IP in the LBR, it is the entry point of the calling function. Similarly, the last source in the LBR buffer was the call site from within the calling function. If the full PEBS record is captured as well, then for functions with limited numbers of arguments on 64-bit OS’s, you can sample both the call counts and the function arguments.

LBRs and Basic Block Execution Counts

Another interesting usage is to use the BR_INST_RETIRED.ALL_BRANCHES event and the LBRs with no filter to evaluate the execution rate of basic blocks. As the LBRs capture all taken branches, all the basic blocks between a branch IP (source) and the previous target in the LBR buffer were executed one time. Thus a simple way to evaluate the basic block execution counts for a given load module is to make a map of the starting locations of every basic block. Then for each sample triggered by the PEBS collection of BR_INST_RETIRED.ALL_BRANCHES, starting from the PEBS address (a target but perhaps for a not taken branch and thus not necessarily in the LBR buffer) and walking backwards through the LBRs until finding an address not corresponding to the load module of interest, count all the basic blocks that were executed. Calling this value “number_of_basic_blocks”, increment the execution counts for all of those blocks by 1/(number_of_basic_blocks). This technique also yields the taken and not taken rates for the active branches. All branch instructions between a source IP and the previous target IP (within the same module) were not taken, while the branches listed in the LBR were taken. This is illustrated in the graphics below.

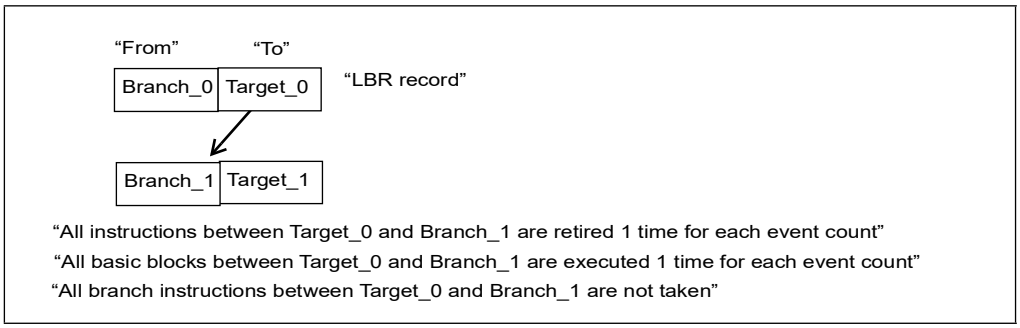


Figure 22-6. LBR Records and Basic Blocks

The 16 sets of LBR records can help rectify the artifact of PEBS samples aggregating disproportionately to certain instructions in the sampling process. The situation of the skewed distribution of the PEBS sample is illustrated in Figure 22-7.

Consider several basic blocks in the flow of normal execution; some basic blocks take twenty cycles to execute, others take two, and shadowing takes ten cycles. Each time an overflow condition occurs, the delay of PEBS being armed is at least ten cycles. Once the PEBS is armed, the PEBS record is captured on the next eventing condition. The skewed distribution of the sampled instruction address using the PEBS record will be skewed, as shown in Figure 22-7. In this conceptual example, every branch is assumed to be taken in these basic blocks.

In the skewed distribution of PEBS samples, the branch IP of the last basic block will be recorded five times as much as the least sampled branch IP address (the second basic block).

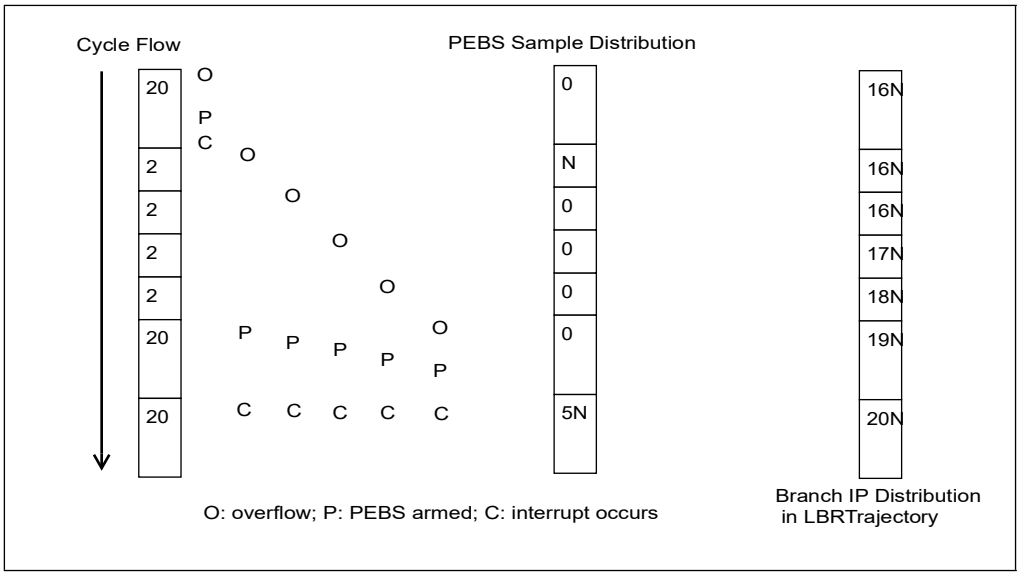


Figure 22-7. Using LBR Records to Rectify Skewed Sample Distribution

This situation where some basic blocks would appear to never get samples and some have many times too many. Weighting each entry by 1/(num of basic blocks in the LBR trajectory), in this example would result in dividing the numbers in the right most table by 16. Thus far more accurate execution counts are achieved ((1.25-> 1.0) * N) in all of the basic blocks, even those that never directly caused a PEBS sample.

As on Intel® Core™2 processors there is a precise instructions retired event that can be used in a wide variety of ways. In addition there are precise events for uops_retired, various SSE instruction classes, FP assists. It should be noted that the FP assist events only detect x87 FP assists, not those involving SSE FP instructions. Detecting all assists will be discussed in the section on the pipeline Frontend.

The instructions retired event has a few special uses. While its distribution is not uniform, the totals are correct. If the values recorded for all the instructions in a basic block are averaged, a measure of the basic block execution count can be extracted. The ratios of basic block executions can be used to estimate loop tripcounts when the counted loop technique discussed above cannot be applied.

The PEBS version (general counter) instructions retired event can further be used to profile OS execution accurately even in the face of STI/CLI semantics, because the PEBS interrupt then occurs after the critical section has completed, but the data was frozen correctly. If the CMask value is set to some very high value and the invert condition is applied, the result is always true, and the event will count core cycles (halted + unhalted).

Consequently both cycles and instructions retired can be accurately profiled. The UOPS_RETIRED.ANY event, which is also precise can also be used to profile Ring 0 execution and really gives a more accurate display of execution. The precise events available for this purpose are listed under event code C0H, C2H, C7H, F7H *at: https://perfmon-events.intel.com/*.

Measuring Core Memory Access Latency

Drilling down performance issues associated with locality or cache coherence issues will require using performance monitoring events. In each processor core, there is a super queue that allocates entries to buffer requests of memory access traffic due to an L2 miss to the uncore sub-system. Table 22-10 lists various performance events available in the core PMU that can drill down performance issues related to L2 misses.

Table 22-10. Core PMU Events to Drill Down L2 Misses

Core PMU Events	Umask	Event Code
OFFCORE_REQUESTS.DEMAND.READ_DATA[1]	01H	B0H
OFFCORE_REQUESTS.DEMAND.READ_CODE[1]	02H	B0H
OFFCORE_REQUESTS.DEMAND.RFO[1]	04H	B0H
OFFCORE_REQUESTS.ANY.READ	08H	B0H
OFFCORE_REQUESTS.ANY.RFO	10H	B0H
OFFCORE_REQUESTS.UNCACHED_MEM	20H	B0H
OFFCORE_REQUESTS.L1D.WRITEBACK	40H	B0H
OFFCORE_REQUESTS.ANY	80H	B0H

NOTES:
1. The *DEMAND* events also include any requests made by the L1D cache hardware prefetchers.

Table 22-11 lists various performance events available in the core PMU that can drill down performance issues related to super queue operation.

Table 22-11. Core PMU Events for Super Queue Operation

Core PMU Events	Umask	Event Code
OFFCORE_REQUESTS_BUFFER_FULL	01H	B2H

Additionally, L2 misses can be drilled down further by data origin attributes and response attributes. The matrix to specify data origin and response type attributes is done by a dedicated MSR OFFCORE_RSP_0 at address 1A6H. See Table 22-12 and Table 22-13.

Table 22-12. Core PMU Event to Drill Down OFFCore Responses

Core PMU Events	OFFCORE_RSP_0 MSR	Umask	Event Code
OFFCORE_RESPONSE	See Table 22-13	01H	B7H

Table 22-13. OFFCORE_RSP_0 MSR Programming

	Position	Description	Note
Request type	0	Demand Data Rd = DCU reads (includes partials, DCU Prefetch)	
	1	Demand RFO = DCU RFOs	
	2	Demand IFetch = IFU Fetches	
	3	Writeback = L2_EVICT/DCUWB	
	4	PF Data Rd = L2 Prefetcher Reads	
	5	PF RFO= L2 Prefetcher RFO	
	6	PF IFetch= L2 Prefetcher Instruction fetches	
	7	Other	Include non-temporal stores
	8	L3_HIT_UNCORE_HIT	exclusive line
	9	L3_HIT_OTHER_CORE_HIT_SNP	clean line
	10	L3_HIT_OTHER_CORE_HITM	modified line
	11	L3_MISS_REMOTE_HIT_SCRUB	Used by multiple cores
	12	L3_MISS_REMOTE_FWD	Clean line used by one core
	13	L3_MISS_REMOTE_DRAM	
	14	L3_MISS_LOCAL_DRAM	
	15	Non-DRAM	Non-DRAM requests

Although Table 22-13 allows 2^16 combinations of setting in MSR_OFFCORE_RSP_0 in theory, it is more useful to consider combining the subsets of 8-bit values to specify “Request type” and “Response type”. The more common 8-bit mask values are listed in Table 22-14.

Table 22-14. Common Request and Response Types for OFFCORE_RSP_0 MSR

Request Type	Mask	Response Type	Mask
ANY_DATA	xx11H	ANY_CACHE_DRAM	7FxxH
ANY_IFETCH	xx44H	ANY_DRAM	60xxH
ANY_REQUEST	xxFFH	ANY_L3_MISS	F8xxH
ANY_RFO	xx22H	ANY_LOCATION	FFxxH
CORE_WB	xx08H	IO	80xxH
DATA_IFETCH	xx77H	L3_HIT_NO_OTHER_CORE	01xxH
DATA_IN	xx33H	L3_OTHER_CORE_HIT	02xxH

Table 22-14. Common Request and Response Types for OFFCORE_RSP_0 MSR (Contd.)

Request Type	Mask	Response Type	Mask
DEMAND_DATA	xx03H	L3_OTHER_CORE_HITM	04xxH
DEMAND_DATA_RD	xx01H	LOCAL_CACHE	07xxH
DEMAND_IFETCH	xx04H	LOCAL_CACHE_DRAM	47xxH
DEMAND_RFO	xx02H	LOCAL_DRAM	40xxH
OTHER[1]	xx80H	REMOTE_CACHE	18xxH
PF_DATA	xx30H	REMOTE_CACHE_DRAM	38xxH
PF_DATA_RD	xx10H	REMOTE_CACHE_HIT	10xxH
PF_IFETCH	xx40H	REMOTE_CACHE_HITM	08xxH
PF_RFO	xx20H	REMOTE-DRAM	20xxH
PREFETCH	xx70H		

NOTES:

1. The PMU may report incorrect counts with setting MSR_OFFCORE_RSP_0 to the value of 4080H. Non-temporal stores to the local DRAM is not reported in the count.

22.4.3.5 Measuring Per-Core Bandwidth

Measuring the bandwidth of all memory traffic for an individual core is complicated, the core PMU and uncore PMU do provide capability to measure the important components of per-core bandwidth.

At the microarchitectural level, there is the buffering of L3 for writebacks/evictions from L2 (similarly to some degree with the non temporal writes). The eviction of modified lines from the L2 causes a write of the line back to the L3. The line in L3 is only written to memory when it is evicted from the L3 some time later (if at all). And L3 is part of the uncore sub-system, not part of the core.

The writebacks to memory due to eviction of modified lines from L3 cannot be associated with an individual core in the uncore PMU logic. The net result of this is that the total write bandwidth for all the cores can be measured with events in the uncore PMU. The read bandwidth and the non-temporal write bandwidth can be measured on a per core basis. In a system populated with two physical processor, the NUMA nature of memory bandwidth implies the measurement for those 2 components has to be divided into bandwidths for the core on a per-socket basis.

The per-socket read bandwidth can be measured with the events:

OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_LOCAL_DRAM.

OFFCORE_RESPONSE_0.DATA_IFETCH.L3_MISS_REMOTE_DRAM.

The total read bandwidth for all sockets can be measured with the event:

OFFCORE_RESPONSE_0.DATA_IFETCH.ANY_DRAM.

The per-socket non-temporal store bandwidth can be measured with the events:

OFFCORE_RESPONSE_0.OTHER.L3_MISS_LOCAL_CACHE_DRAM.

OFFCORE_RESPONSE_0.OTHER.L3_MISS_REMOTE_DRAM.

The total non-temporal store bandwidth can be measured with the event:

OFFCORE_RESPONSE_0.OTHER.ANY.CACHE_DRAM.

The use of “CACHE_DRAM” encoding is to work around the defect in the footnote of Table 22-14. Note that none of the above includes the bandwidth associated with writebacks of modified cacheable lines.

22.4.3.6 Miscellaneous L1 and L2 Events for Cache Misses

In addition to the OFFCORE_RESPONSE_0 event and the precise events that will be discussed later, there are several other events that can be used as well. There are additional events that can be used to supplement the offcore_response_0 events, because the offcore_response_0 event code is supported on counter 0 only.

L2 misses can also be counted with the architecturally defined event LONGEST_LAT_CACHE_ACCESS, however as this event also includes requests due to the L1D and L2 hardware prefetchers, its utility may be limited. Some of the L2 access events can be used for both drilling down L2 accesses and L2 misses by type, in addition to the OFFCORE_REQUESTS events discussed earlier. The L2_RQSTS and L2_DATA_RQSTS events can be used to discern assorted access types. In all of the L2 access events the designation PREFETCH only refers to the L2 hardware prefetch. The designation DEMAND includes loads and requests due to the L1D hardware prefetchers.

The L2_LINES_IN and L2_LINES_OUT events have been arranged slightly differently than the equivalent events on Intel® Core™2 processors. The L2_LINES_OUT event can now be used to decompose the evicted lines by clean and dirty (i.e. a Writeback) and whether they were evicted by an L1D request or an L2 HW prefetch.

The event L2_TRANSACTIONS counts all interactions with the L2.

Writes and locked writes are counted with a combined event, L2_WRITE.

The details of the numerous derivatives of L2_RQSTS, L2_DATA_RQSTS, L2_LINES_IN, L2_LINES_OUT, L2_TRANSACTIONS, L2_WRITE, can be found under event codes24H, 26H, F1H, F2H, F0H, and 27H *at: https://perfmon-events.intel.com/*.

22.4.3.7 TLB Misses

The next largest set of memory access delays are associated with the TLBs when linear-to-physical address translation is mapped with a finite number of entries in the TLBs. A miss in the first level TLBs results in a very small penalty that can usually be hidden by the OOO execution and compiler's scheduling. A miss in the shared TLB results in the Page Walker being invoked and this penalty can be noticeable in the execution.

The (non-PEBS) TLB miss events break down into three sets:

- DTLB misses and its derivatives are programmed with event code 49H.
- Load DTLB misses and its derivatives are programmed with event code 08H.
- ITLB misses and its derivatives are programmed with event code 85H.

Store DTLB misses can be evaluated from the difference of the DTLB misses and the Load DTLB misses. Each then has a set of sub events programmed with the Umask value. The Umask details of the numerous derivatives of the above events are listed *at: https://perfmon-events.intel.com/*.

22.4.3.8 L1 Data Cache

There are PMU events that can be used to analyze L1 data cache operations. These events can only be counted with the first 2 of the 4 general counters, i.e. IA32_PMC0 and IA32_PMC1. Most of the L1D events are self explanatory.

The total number of references to the L1D can be counted with L1D_ALL_REF, either just cacheable references or all. The cacheable references can be divided into loads and stores with L1D_CACHE_LOAD and L1D_CACHE.STORE. These events are further subdivided by MESI states through their Umask values, with the I state references indicating the cache misses.

The evictions of modified lines in the L1D result in writebacks to the L2. These are counted with the L1D_WB_L2 events. The Umask values break these down by the MESI state of the version of the line in the L2.

The locked references can be counted also with the L1D_CACHE_LOCK events. Again these are broken down by MES states for the lines in L1D.

The total number of lines brought into L1D, the number that arrived in an M state and the number of modified lines that get evicted due to receiving a snoop are counted with the L1D event and its Umask variations.

The L1D events are listed under event codes28H, 40H, 41H, 42H, 43H, 48H, 4EH, 51H, 52H, 53H, 80H, and 83H *at: https://perfmon-events.intel.com/*.

There are few cases of loads not being able to forward from active store buffers. The predominant situations have to do with larger loads overlapping smaller stores. There is not event that detects when this occurs. There is also a “false store forwarding” case where the addresses only match in the lower 12 address bits. This is sometimes referred to as 4K aliasing. This can be detected with the event “PARTIAL_ADDRESS_ALIAS“ which has event code 07H and Umask 01H.

22.4.4 FRONTEND MONITORING EVENTS

Branch misprediction effects can sometimes be reduced through code changes and enhanced inlining. Most other Frontend performance limitations have to be dealt with by the code generation. The analysis of such issues is mostly of use by compiler developers.

22.4.4.1 Branch Mispredictions

In addition to branch retired events that was discussed in conjunction with PEBS in Section 22.4.3.3. These are enhanced by use of the LBR to identify the branch location to go along with the target location captured in the PEBS buffer. Aside from those usage, many other PMU events (event code E6, E5, E0, 68, 69) associated with branch predictions are more relevant to hardware design than performance tuning.

Branch mispredictions are not in and of themselves an indication of a performance bottleneck. They have to be associated with dispatch stalls and the instruction starvation condition, UOPS_ISSUED:C1:I1 – RESOURCE_STALLS.ANY. Such stalls are likely to be associated with ICache misses and ITLB misses. The precise ITLB miss event can be useful for such issues. The ICache and ITLB miss events are listed under event code 80H, 81H, 82H, 85H, AEH.

22.4.4.2 Frontend Code Generation Metrics

The remaining Frontend events are mostly of use in identifying when details of the code generation interact poorly with the instructions decoding and µop issue to the OOO engine. Examples are length changing prefix issues associated with the use of 16 bit immediates, rob read port stalls, instruction alignment interfering with the loop detection and instruction decoding bandwidth limitations. The activity of the LSD is monitored using CMASK values on a signal monitoring activity. Some of these events are listed under event code 17H, 18H, 1EH, 1FH, 87H, A6H, A8H, D0H, D2H *at:*

https://perfmon-events.intel.com/.

Some instructions (FSIN, FCOS, and other transcendental instructions) are decoded with the assistance of MS-ROM. Frequent occurrences of instructions that required assistance of MS-ROM to decode complex µop flows are opportunity to improve instruction selection to reduce such occurrences. The UOPS_DECODED.MS event can be used to identify code regions that could benefit from better instruction selection.

Other situations that can trigger this event are due to FP assists, like performing a numeric operation on denormalized FP values or QNaNs. In such cases the penalty is essentially the µops required for the assist plus the pipeline clearing required to ensure the correct state.

Consequently this situation has a very clear signature consisting of MACHINE_CLEAR.CYCLES and µops being inserted by the microcode sequencer, UOPS_DECODED.MS. The execution penalty being the sum of these two contributions. The event codes for these are listed under D1H and C3H.

22.4.5 UNCORE PERFORMANCE MONITORING EVENTS

The uncore sub-system includes the L3, IMC and Intel QPI units in the diagram shown in Figure 22-4. Within the uncore sub-system, the uncore PMU consists of eight general-purpose counters and one fixed counter. The fixed counter in uncore monitors the unhalted clock cycles in the uncore clock domain, which runs at a different frequency than the core.

The uncore cannot by itself generate a PMI interrupt. While the core PMU can raise PMI at a per-logical-processor specificity, the uncore PMU can cause PMI at a per-core specificity using the interrupt hardware in the processor core. When an uncore counter overflows, a bit pattern is used to specify which cores should be signaled to raise a PMI. The uncore PMU is unaware of the core, Processor ID or Thread ID that caused the event that overflowed a counter. Consequently the most reasonable approach for sampling on uncore events is to raise a PMI on all the logical processors in the package.

There are a wide variety of events that monitor queue occupancies and inserts. There are others that count cacheline transfers, dram paging policy statistics, snoop types and responses, and so on. The uncore is the only place the total bandwidth to memory can be measured. This will be discussed explicitly after all the uncore components and their events are described.

22.4.5.1 Global Queue Occupancy

Each processor core has a super queue that buffers requests of memory access traffic due to an L2 miss. The uncore has a global queue (GQ) to service transaction requests from the processor cores and buffers data traffic that arrive from L3, IMC, or Intel QPI links.

Within the GQ, there are 3 “trackers” in the GQ for three types of transactions:

- On-package read requests, its tracker queue has 32 entries.
- On-package writeback requests, its tracker queue has 16 entries.
- Requests that arrive from a “peer”, its tracker queue has 12 entries.

A “peer” refers to any requests coming from the Intel® QuickPath Interconnect.

The occupancies, inserts, cycles full and cycles not empty for all three trackers can be monitored. Further as load requests go through a series of stages the occupancy and inserts associated with the stages can also be monitored, enabling a “cycle accounting” breakdown of the uncore memory accesses due to loads.

When a uncore counter is first programmed to monitor a queue occupancy, for any of the uncore queues, the queue must first be emptied. This is accomplished by the driver of the monitoring software tool issuing a bus lock. This only needs to be done when the counter is first programmed. From that point on the counter will correctly reflect the state of the queue, so it can be repeatedly sampled for example without another bus lock being issued.

The uncore events that monitor GQ allocation (UNC_GQ_ALLOC) and GQ tracker occupancy (UNC_GQ_TRACKER_OCCUP) are listed under the event code 03H and 02H *at: https://perfmon-events.intel.com/*. The selection between the three trackers is specified from the Umask value. The mnemonic of these derivative events use the notation: “RT” signifying the read tracker, “WT”, the write tracker and “PPT” the peer probe tracker.

Latency can measured by the average duration of the queue occupancy, if the occupancy stops as soon as the data has been delivered. Thus the ratio of UNC_GQ_TRACKER_OCCUP.X/UNC_GQ_ALLOC.X measures an average duration of queue occupancy, where 'X' represents a specific Umask value. The total occupancy period of the read tracker as measured by:

Total Read Period = UNC_GQ_TRACKER_OCCUP.RT/UNC_GQ_ALLOC.RT

Is longer than the data delivery latency due to it including time for extra bookkeeping and cleanup. The measurement:

LLC response Latency = UNC_GQ_TRACKER_OCCUP.RT_TO_LLC_RESP / UNC_GQ_ALLOC.RT_TO_LLC_RESP

is essentially a constant. It does not include the total time to snoop and retrieve a modified line from another core for example, just the time to scan the L3 and see if the line is or is not present in this socket.

An overall latency for an L3 hit is the weighted average of three terms:

- The latency of a simple hit, where the line has only been used by the core making the request.
- The latencies for accessing clean lines by multiple cores.
- The latencies for accessing dirty lines that have been accessed by multiple cores.

These three components of the L3 hit for loads can be decomposed using the derivative events of OFFCORE_RESPONSE:

- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_NO_OTHER_CORE.
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HIT.
- OFFCORE_RESPONSE_0.DEMAND_DATA.L3_HIT_OTHER_CORE_HITM.

The event OFFCORE_RESPONSE_0.DEMAND_DATA.LOCAL_CACHE should be used as the denominator to obtain latencies. The individual latencies could have to be measured with microbenchmarks, but the use of the precise latency event will be far more effective as any bandwidth loading effects will be included.

The L3 miss component is the weighted average over three terms:

- The latencies of L3 hits in a cache on another socket (this is described in the previous paragraph).
- The latencies to local DRAM.
- The latencies to remote DRAM.

The local dram access and the remote socket access can be decomposed with more uncore events.

Miss to fill latency = UNC_GQ_TRACKER_OCCUP.RT_LLC_MISS / UNC_GQ_ALLOC.RT_LLC_MISS

The uncore GQ events using Umask value associated with *RTID* mnemonic allow the monitoring of a sub component of the Miss to fill latency associated with the communications between the GQ and the QHL.

There are uncore PMU events which monitor cycles when the three trackers are not empty (>= 1 entry) or full. These events are listed under the event code 00H and 01H *at:* https://perfmon-events.intel.com/.

Because the uncore PMU generally does not differentiate which processor core causes a particular eventing condition, the technique of dividing the latencies by the average queue occupancy to determine a penalty does not work for the uncore. Overlapping entries from different cores do not result in overlapping penalties and thus a reduction in stalled cycles. Each core suffers the full latency independently.

To evaluate the correction on a per-core basis, the number of cycles is required for an entry from the core in question. A *NOT_EMPTY_CORE_N type event is required, however, there is no such event. Consequently, in the cycle decomposition one must use the full latency for the estimate of the penalty. As has been stated before it is best to use the PEBS latency event as the data sources are also collected with the latency for the individual sample.

The individual components of the read tracker, discussed above, can also be monitored as busy or full by setting the CMask value to 1 or 32 and applying it to the assorted read tracker occupancy events.

Table 22-15. Uncore PMU Events for Occupancy Cycles

Uncore PMU Events	CMask	Umask	Event Code
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_FULL	32	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_FULL	32	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_FULL	32	08H	02H
UNC_GQ_TRACKER_OCCUP.RT_L3_MISS_BUSY	1	02H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_L3_RESP_BUSY	1	04H	02H
UNC_GQ_TRACKER_OCCUP.RT_TO_RTID_ACCQUIRED_BUSY	1	08H	02H

22.4.5.2 Global Queue Port Events

The GQ data buffer traffic controls the flow of data to and from different sub-systems via separate ports:

- Core traffic: two ports handles data traffic, each port dedicated to a pair of processor cores.
- L3 traffic: one port service L3 data traffic.
- Intel QPI traffic: one service traffic to QPI logic.
- IMC traffic: one service data traffic to integrated memory controller.

The ports for L3 and core traffic transfer a fixed number of bits per cycle. However the Intel® QuickPath Interconnect protocols can result in either 8 or 16 bytes being transferred on the read Intel QPI and IMC ports. Consequently these events cannot be used to measure total data transfers and bandwidths.

The uncore PMU events that can distinguish traffic flow are listed under the event code 04H and 05H *at: https://perfmon-events.intel.com/*.

22.4.5.3 Global Queue Snoop Events

Cacheline requests from the cores or from a remote package or the I/O Hub are handled by the GQ. When the uncore receives a cacheline request from one of the cores, the GQ first checks the L3 to see if the line is on the package. Because the L3 is inclusive, this answer can be quickly ascertained. If the line is in the L3 and was owned by the requesting core, data can be returned to the core from the L3 directly. If the line is being used by multiple cores, the GQ will snoop the other cores to see if there is a modified copy. If so the L3 is updated and the line is sent to the requesting core.

In the event of an L3 miss, the GQ must send out requests to the local memory controller (or over the Intel QPI links) for the line. A request through the Intel QPI to a remote L3 (or remote DRAM) must be made if data exists in a remote L3 or does not exist in local DRAM. As each physical package has its own local integrated memory controller the GQ must identify the “home” location of the requested cacheline from the physical address. If the address identifies home as being on the local package then the GQ makes a simultaneous request to the local memory controller. If home is identified as belonging to the remote package, the request sent over the Intel QPI will also access the remote IMC.

The GQ handles the snoop responses for the cacheline requests that come in from the Intel® QuickPath Interconnect. These snoop traffic correspond to the queue entries in the peer probe tracker.

The snoop responses are divided into requests for locally homed data and remotely homed data. If the line is in a modified state and the GQ is responding to a read request, the line also must be written back to memory. This would be a wasted effort for a response to a RFO as the line will just be modified again, so no Writeback is done for RFOs.

The snoop responses of local home events that can be monitored by an uncore PMU are listed under event code 06H *at: https://perfmon-events.intel.com/*. The snoop responses of remotely home events are listed under event code 07H.

Some related events count the MESI transitions in response to snoops from other caching agents (processors or IOH). Some of these rely on programming MSR so they can only be measured one at a time, as there is only one MSR. The Intel performance tools will schedule this correctly by restricting these events to a single general uncore counter.

22.4.5.4 L3 Events

Although the number of L3 hits and misses can be determined from the GQ tracker allocation events, Several uncore PMU event is simpler to use. They are listed under event code 08H and 09H in the uncore event list *at: https://perfmon-events.intel.com/*.

The MESI states breakdown of lines allocated and victimized can also be monitored with LINES_IN, LINES_OUT events in the uncore using event code 0AH and 0BH.

22.4.6 INTEL QUICKPATH INTERCONNECT HOME LOGIC (QHL)

When a data misses L3 and causing the GQ of the uncore to send out a transaction request, the Intel QPI fabric will fulfill the request either from the local DRAM controller or from a remote DRAM controller in another physical package. The GQ must identify the “home” location of the requested cacheline from the physical address. If the address identifies home as being on the local package then the GQ makes a simultaneous request to the local memory controller, the Integrated memory controller (IMC). If home is identified as belonging to the remote package, the request is sent to the Intel QPI first and then to access the remote IMC.

The Intel QPI logic and IMC are distinct units in the uncore sub-system. The Intel QPI logic distinguish the local IMC relative to remote IMC using the concept of “caching agent” and “home agent“. Specifically, the Intel QPI protocol considers each socket as having a “caching agent”: and a “home agent”:

- Caching Agent is the GQ and L3 in the uncore (or an IOH if present).
- Home Agent is the IMC.

An L3 miss result in simultaneous queries for the line from all the Caching Agents and the Home agent (wherever it is).

QHL requests can be superseded when another source can supply the required line more quickly. L3 misses to locally homed lines, due to on package requests, are simultaneously directed to the QHL and Intel QPI. If a remote caching agent supplies the line first then the request to the QHL is sent a signal that the transaction is complete. If the remote caching agent returns a modified line in response to a read request then the data in dram must be updated with a writeback of the new version of the line.

There is a similar flow of control signals when the Intel QPI simultaneously sends a snoop request for a locally homed line to both the GQ and the QHL. If the L3 has the line, the QHL must be signaled that the transaction was completely by the L3/GQ. If the line in L3 (or the cores) was modified and the snoop request from the remote package was for a load, then a writeback must be completed by the QHL and the QHL forwards the line to the Intel QPI to complete the transaction.

Uncore PMU provides events for monitoring these cacheline access and writeback traffic in the uncore by using the QHL opcode matching capability. The uncore PMU event that uses the opcode matching capability is listed under event code 35H. Several of the more useful setting to program QHL opcode matching is shown in Table 22-16.

Table 22-16. Common QHL Opcode Matching Facility Programming

Load Latency Precise Events	MSR 0x396	Umask	Event Code
UNC_ADDR_OPCODE_MATCH.IOH.NONE	0	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.IOH.RSPIWB	40001D00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.NONE	0	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDI	40001900_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPFWDS	40001A00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.REMOTE.RSPIWB	40001D00_00000000	2H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.NONE	0	4H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDI	40001900_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPFWDS	40001A00_00000000	1H	35H
UNC_ADDR_OPCODE_MATCH.LOCAL.RSPIWB	40001D00_00000000	1H	35H

These predefined opcode match encodings can be used to monitor HITM accesses. It is the only event that allows profiling the code requesting HITM transfers.

The diagrams Figure 22-8 through Figure 22-15 show a series of Intel QPI protocol exchanges associated with Data Reads and Reads for Ownership (RFO), after an L3 miss, under a variety of combinations of the local home of the cacheline, and the MESI state in the remote cache. Of particular note are the cases where the data comes from the remote QHL even when the data was in the remote L3. These are the Read Data with the remote L3 having the line in an M state.

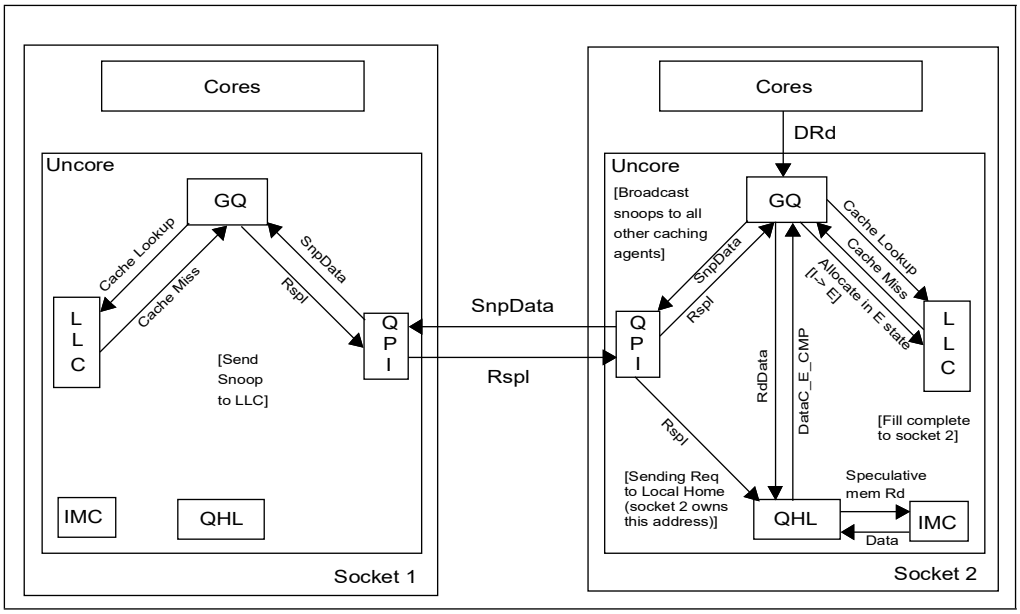


Figure 22-8. RdData Request after LLC Miss to Local Home (Clean Rsp)

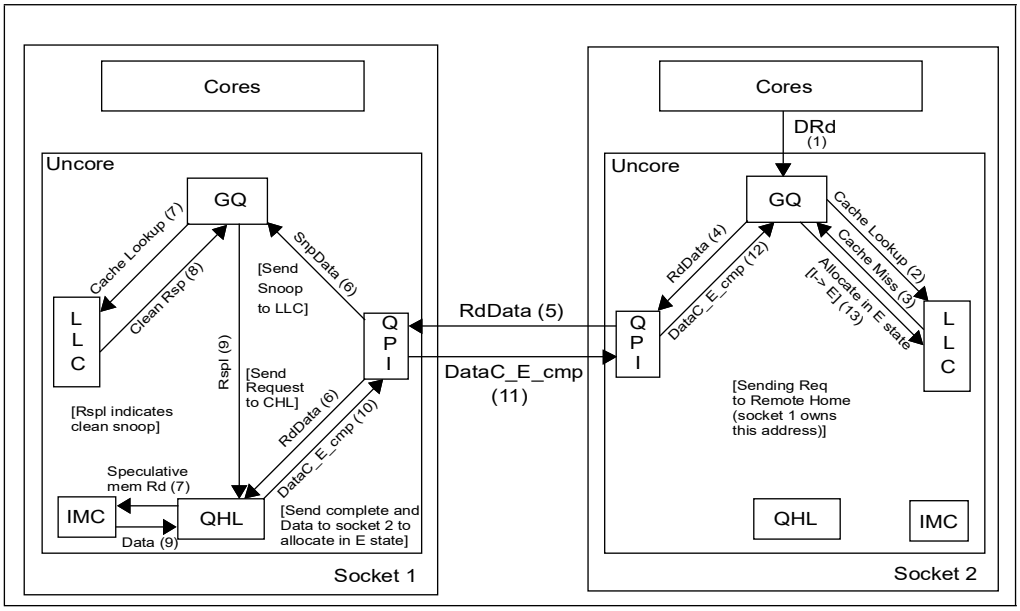


Figure 22-9. RdData Request after LLC Miss to Remote Home (Clean Rsp)

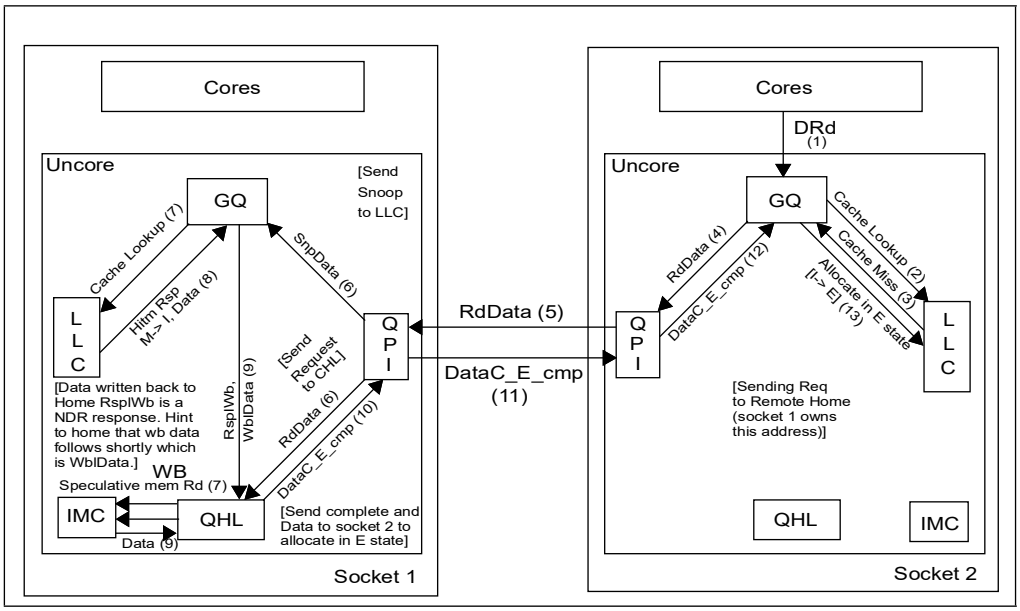


Figure 22-10. RdData Request after LLC Miss to Remote Home (Hitm Response)

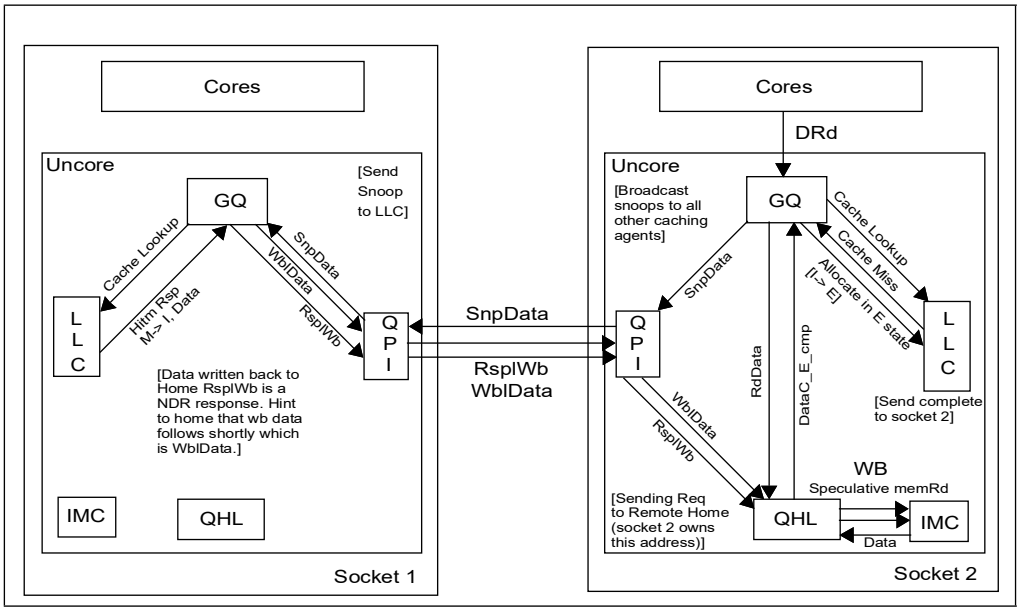


Figure 22-11. RdData Request after LLC Miss to Local Home (Hitm Response)

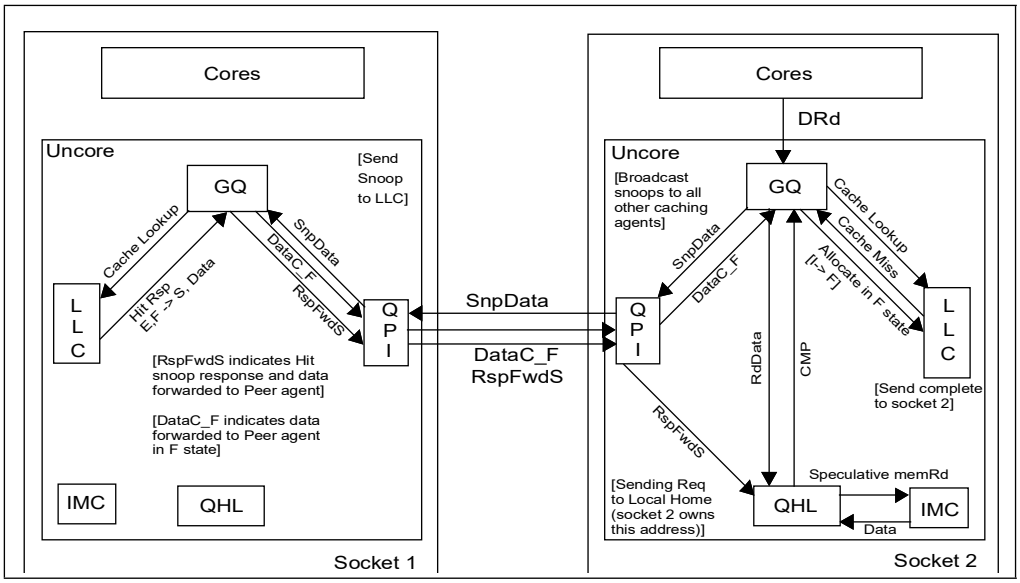


Figure 22-12. RdData Request after LLC Miss to Local Home (Hit Response)

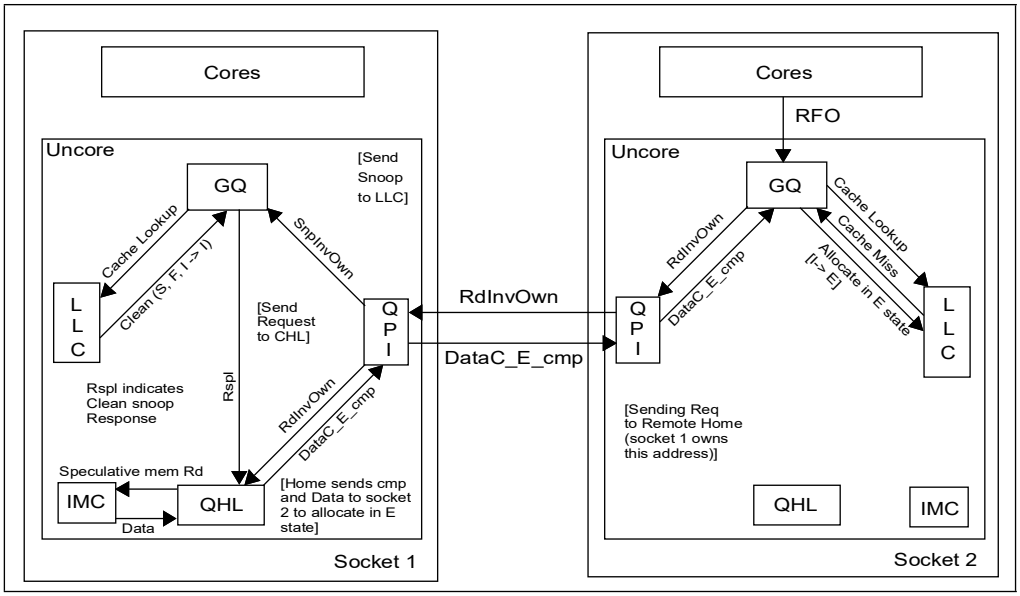


Figure 22-13. RdInvOwn Request after LLC Miss to Remote Home (Clean Res)

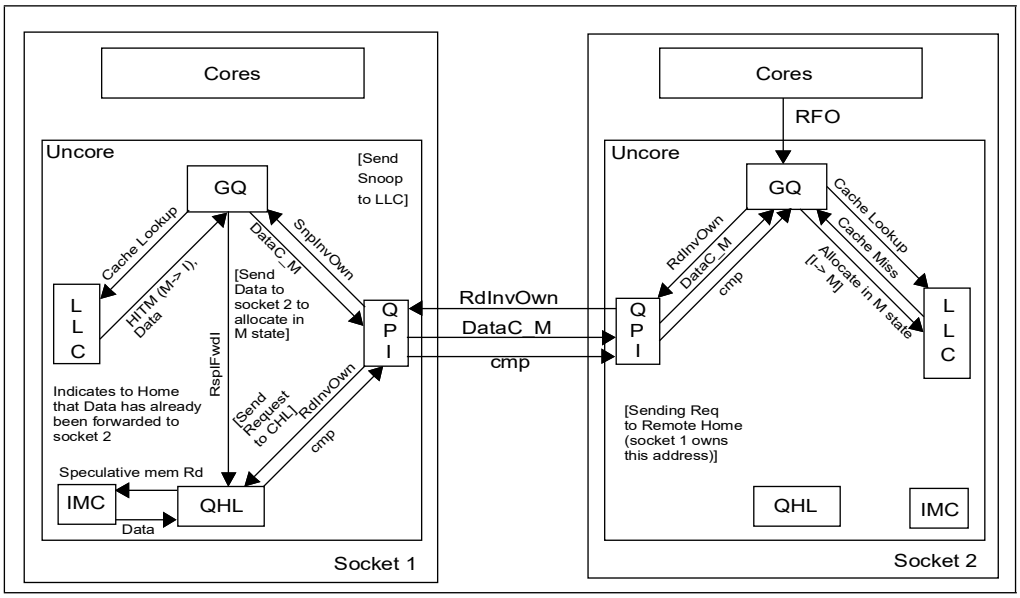


Figure 22-14. RdInvOwn Request after LLC Miss to Remote Home (Hitm Res)

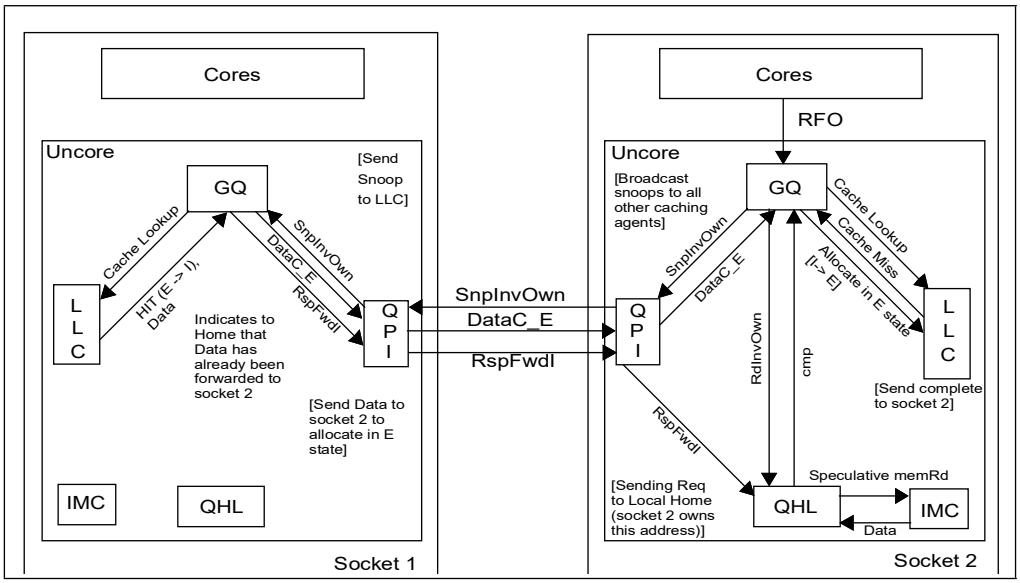


Figure 22-15. RdInvOwn Request after LLC Miss to Local Home (Hit Res)

Whether the line is locally or remotely “homed” it has to be written back to dram before the originating GQ receives the line, so it always appears to come from a QHL. The RFO does not do this. However, when responding to a remote RFO (SnpInvOwn) and the line is in an S or F state, the cacheline gets invalidated and the line is sent from the QHL. The point is that the data source might not always be so obvious.

22.4.7 MEASURING BANDWIDTH FROM THE UNCORE

Read bandwidth can be measured on a per core basis using events like OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM and OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM. The total bandwidth includes writes and these cannot be monitored from the core as they are mostly caused by evictions of modified lines in the L3. Thus a line used and modified by one core can end up being written back to dram when it is evicted due to a read on another core doing some completely unrelated task. Modified cached lines and writebacks of uncached lines (e.g. written with non temporal streaming stores) are handled differently in the uncore and their writebacks increment various events in different ways.

All full lines written to DRAM are counted by the UNC_IMC_WRITES.FULL.* events. This includes the writebacks of modified cached lines and the writes of uncached lines, for example generated by non-temporal SSE stores. The uncached line writebacks from a remote socket will be counted by UNC_QHL_REQUESTS.REMOTE_WRITES. The uncached writebacks from the local cores are not counted by UNC_QHL_REQUESTS.LOCAL_WRITES, as this event only counts writebacks of locally cached lines.

The UNC_IMC_NORMAL_READS.* events only count the reads. The UNC_QHL_REQUESTS.LOCAL_READS and the UNC_QHL_REQUESTS.REMOTE_READS count the reads and the “InvtoE” transactions, which are issued for the uncacheable writes, eg USWC/UC writes. This allows the evaluation of the uncacheable writes, by computing the difference of UNC_QHL_REQUESTS.LOCAL_READS +

UNC_QHL_REQUESTS.REMOTE_READS – UNC_IMC_NORMAL_READS.ANY.

These uncore PMU events that are useful for bandwidth evaluation are listed under event code 20H, 2CH, 2FH *at: https://perfmon-events.intel.com/*.

22.5 PERFORMANCE TUNING TECHNIQUES FOR SANDY BRIDGE MICROARCHITECTURE

This section covers various performance tuning techniques using performance monitoring events. Some techniques can be adapted in general to other microarchitectures, most of the performance events are specific to Sandy Bridge microarchitecture.

22.5.1 CORRELATING PERFORMANCE BOTTLENECK TO SOURCE LOCATION

Performance analysis tools often sample events to identify hot spots of instruction pointer addresses to help programmers identify source locations of potential performance bottlenecks.

The sampling technique requires a service routine to respond to the performance monitoring interrupt (PMI) generated from an overflow condition of the performance counter. There is a finite delay between the performance monitoring event detection of the eventing condition relative to the capture of the instruction pointer address. This is known as “skid“. In other words, the event skid is the distance between the instruction or instructions that caused the issue and the instruction where the event is tagged. There are a few things to note in general on skid:

- Precise events have a defined event skid of 1 instruction to the next instruction retired. In the case when the offending instruction is a branch, the event is tagged with the branch target, which can be separated from the branch instruction. Thus sampling with precise events is likely to have less noise in pin-pointing source locations of bottlenecks.
- Using a performance event with eventing condition that carries a larger performance impact generally has a shorter skid and vice versa. The following examples illustrate this rule:
  - A store forward block issue can cause a penalty of more than 10 cycles. Sampling a store forward block event almost always tags to the next couple of instructions after the blocked load.

  - On the other hand, sampling loads that forwarded successfully with no penalty will have much larger skids, and less helpful for performance tuning.

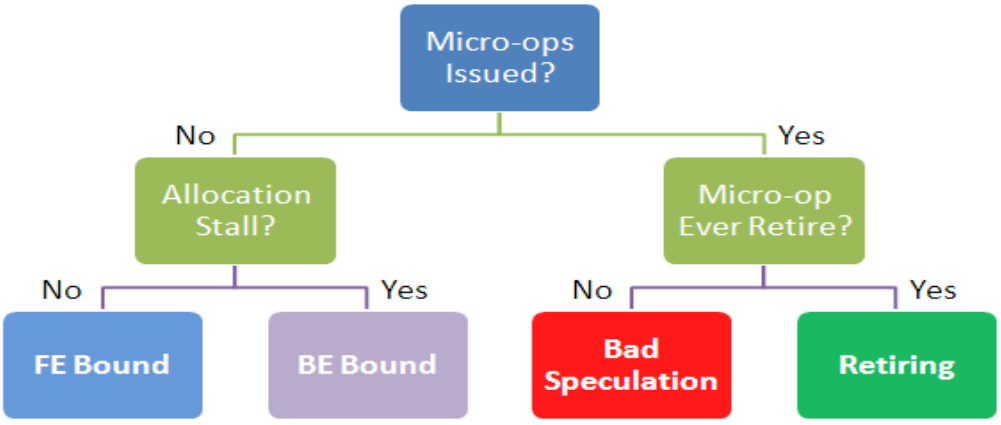
- The closer the eventing condition is to the retirement of the instruction, the shorter the skid. The events in the Frontend of the pipeline tend to tag to instructions further from the responsible instruction than events that are taken at execution or retirement.
- Cycles counted with the event CPU_CLK_UNHALTED.THREAD often tag in greater counts on the instruction after larger bottlenecks in the pipeline. If cycles are accumulated on an instruction this is probably due to a bottleneck on the instruction at the previous instruction.
- It is very difficult to determine the source of issues with a low cost that occur in the Frontend. Frontend events can also skid to IPs that precede the actual instructions that are causing the issue.

22.5.2 HIERARCHICAL TOP-DOWN PERFORMANCE CHARACTERIZATION METHODOLOGY AND LOCATING PERFORMANCE BOTTLENECKS

Sandy Bridge microarchitecture has introduced several performance events which help narrow down which portion of the microarhcitecture pipeline is stalled. This starts with a hierarchical approach to characterize a workload of where CPU cycles are spent in the microarchitecture pipelines. At the top level, there are four areas to attribute CPU cycles; these are described below. To determine what portion of the pipeline is stalled, the technique looks at a buffer that queues the micro-ops supplied by the front end and feeds the out-of-order back end. This buffer is called the micro-op queue. From the micro-op queue viewpoint, there may be four different types of stalls:

- Front end stalls - The front end is delivering less than four micro-ops per cycle when the back end of the pipeline is requesting micro-ops. When these stalls happen, the rename/allocate part of the OOO engine will starved. Thus, execution is said to be front end bound.
- Back end stalls – No micro-ops are being delivered from the micro-op queue due to lack of required resources for accepting more micro-ops in the back end of the pipeline. When these stalls happen, execution is said to be back end bound.
- Bad speculation - The pipeline performs speculative execution of instructions that never successfully retire. The most common case is a branch misprediction where the pipeline predicts a branch target to keep the pipeline full instead of waiting for the branch to execute. If the processor prediction is incorrect it has to flush the pipeline without retiring the speculated instructions.
- Retiring – The micro-op queue delivers micro-ops that eventually retire. In the common case, the micro-ops originate from the program code. One exception is with assists where the microcode sequencer generates micro-ops to deal with issues in the pipeline.

The following figure illustrates how the execution opportunities are logically divided.



It is possible to estimate the amount of execution slots spent in each category using the following formulas in conjunction with core PMU performance events in Sandy Bridge microarchitecture:

%FE_Bound =

100 * (IDQ_UOPS_NOT_DELIVERED.CORE / N);

%Bad_Speculation =

100 * ((UOPS_ISSUED.ANY – UOPS_RETIRED.RETIRE_SLOTS + 4 * INT_MISC.RECOVERY_CYCLES) / N);

%Retiring = 100 * (UOPS_RETIRED.RETIRE_SLOTS/ N);

%BE_Bound = 100 * (1 – (FE_Bound + Retiring + Bad_Speculation));

N represents total execution slots opportunities. Execution opportunities are the number of cycles multiplied by four.

N = 4*CPU_CLK_UNHALTED.THREAD

The following sections explain the source for penalty cycles in three categories: back end stalls, Frontend stalls and bad speculation. They use formulas that can be applied to process, module, function, and instruction granularity.

22.5.2.1 Back End Bound Characterization

Once the %BE_Bound metric raises concern, a user may need to drill down to the next level of possible issues in the back end. Our methodology examines back end stalls based on execution unit occupation at every cycle. Naturally, optimal performance may be achieved when all execution resources are kept busy. Currently, this methodology splits **back end bound** issues into two categories: **memory bound** and **core bound**.

“Memory bound” corresponds to stalls related to the memory subsystem. For example, cache misses may eventually cause execution starvation. On the other hand, “core bound” which corresponds to stalls due to either the Execution- or OOO-clusters, is a bit trickier. These stalls can manifest either with execution starvation or non-optimal execution ports utilization. For example, a long latency divide operation may serialize the execution causing execution starvation for some period, while pressure on an execution port that serves specific types of µops, might manifest as small number of ports utilized in a cycle.

Use performance monitoring events at the execution units to calculate:

%BE_Bound_at_EXE =

(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.THREAD:c1 - UOPS_EXECUTED.THREAD:c2) / CLOCKS

CYCLE_ACTIVITY.CYCLES_NO_EXECUTE counts complete starvation cycles where no µop is executed whatsoever.

UOPS_EXECUTED.THREAD:c1 and UOPS_EXECUTED.THREAD:c2 count cycles where at least 1- and 2-µops were executed in a cycle, respectively. Hence the event count difference measures the cycles when the OOO back end could execute only one µop.

The **%BE_Bound_at_EXE** metric is counted at execution unit pipestages so the number would not match the Backend_Bound ratio which is done at the allocation stage. However, redundancy is good here as one can use both counters to confirm the execution is indeed back end bound (both should be high).

22.5.2.2 Core Bound Characterization

A “back end bound” workload can be identified as “core bound” by the following metric:

%Core_Bound = %Backend_Bound_at_EXE - %Memory_Bound

The metric “%Memory_Bound” is described in Section 22.5.2.3. Once a workload is identified as “core bound”, the user may want to drill down into OOO or Execution related issues through their transitional targeted performance counter, like, for example, execution ports pressure, or use of FP-chained long-latency arithmetic operations.

22.5.2.3 Memory Bound Characterization

More primitive methods of characterizing performance issues in the memory pipeline tend to use naïve calculations to estimate the penalty of memory stalls. Usually the number of misses to a given cache level access is multiplied by a predefined latency for that cache level per the CPU specifications, to get an estimation for the penalty. While this might work for an in-order processor, it often over-estimates the contribution of memory accesses on CPU cycles for highly out-of-order processors, because memory accesses tend to overlap and the scheduler manages to hide a good portion of the latency. The scheduler might be able to hide some of the memory access stalls by keeping the execution stalls busy with µops that do not require the memory access data. Thus penalty for a memory access is when the scheduler has nothing more ready to dispatch and the execution units get starved as a result. It is likely that further µops are either waiting for memory access data, or depend on other non-dispatched µops.

In Ivy Bridge microarchitecture, a new performance monitoring event “CYCLE_ACTIVITY.STALLS_LDM_PENDING” is provided to estimate the exposure of memory accesses. It is used to define the “**memory bound**” metric. This event measures the cycles when there is a non-completed in-flight memory demand load coincident with execution starvation. Note that only demand load operations are accounted for, as µops do not typically wait for (direct) completion of stores or HW prefetches:

%Memory_Bound = CYCLE_ACTIVITY.STALLS_LDM_PENDING / CLOCKS

If a workload is memory bound, it is possible to further characterize its performance characteristic with respect to the contribution of the cache hierarchy and DRAM system memory.

L1 cache has typically the shortest latency which is comparable to ALU units' stalls that are the shortest among all stalls. Yet in certain cases, like loads blocked on older stores, a load might suffer high latency while eventually being satisfied by the L1. There are no fill-buffers allocated for L1 hits; use the LDM stalls sub-event instead because it accounts for any non-completed load.

%L1 Bound = (CYCLE_ACTIVITY.STALLS_LDM_PENDING -CYCLE_ACTIVITY.STALLS_L1D_PENDING)/ CLOCKS

As explained above, L2 Bound is detected as:

%L2 Bound =(CYCLE_ACTIVITY.STALLS_L1D_PENDING - CYCLE_ACTIVITY.STALLS_L2_PENDING)/ CLOCKS

In principle, L3 Bound can be calculated similarly by subtracting out the L3 miss contribution. However an equivalent event to measure L3_PENDING is unavailable. Nevertheless, it is possible to infer an estimate using L3_HIT and L3_MISS load count events in conjunction with a correction factor. This estimation could be tolerated as the latencies are longer on L3 and Memory. The correction factor MEM_L3_WEIGHT is approximately the external memory to L3 cache latency ratio. A factor of 7 can be used for the third generation Intel Core processor family. Note this correction factor has some dependency on CPU and Memory frequencies.

%L3 Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Hit_fraction / CLOCKS

Where L3_Hit_fraction is:

MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ MEM_L3_WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

To estimate the exposure of DRAM traffic on third generation Intel Core processors, the remainder of L2_PENDING is used for MEM Bound:

%MEM Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING * L3_Miss_fraction / CLOCKS

Where L3_Miss_fraction is:

WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS / (MEM_LOAD_UOPS_RETIRED.LLC_HIT+ WEIGHT*MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)

Sometimes it is meaningful to refer to all memory stalls outside the core as Uncore Bound:

%Uncore Bound = CYCLE_ACTIVITY.STALLS_L2_PENDING / CLOCKS

22.5.3 BACK END STALLS

Back end stalls have two main sources: memory sub-system stalls and execution stalls. As a first step to understanding the source of back end stalls, use the resource stall event.

Before putting micro-ops into the scheduler, the rename stage has to have certain resources allocated. When an application encounters a significant bottleneck at the back end of the pipeline, it runs out of these resources as the pipeline backs up. The RESOURCE_STALLS event tracks stall cycles when a resource could not be allocated. The event breaks up each resource into a separate sub-event so you can track which resource is unavailable for allocation. Counting these events can help identifying the reason for issues in the back end of the pipeline.

The resource stall ratios described below can be accomplished at process, module, function and even instruction granularities with the cycles, counted by CPU_CLK_UNHALTED.THREAD, representing the penalty tagged at the same granularity.

Usages of Specific Events

RESOURCE_STALLS.ANY - Counts stall cycles that the rename stage is unable to put micro-ops into the scheduler, due to lack of resources that have to be allocated at this stage. The event skid tends to be low since it is close to the retirement of the blocking instruction. This event accounts for all stalls counted by other RESOURCE_STALL sub events and also includes the sub-events of RESOURCE_STALLS2. If this ratio is high, count the included sub-events to get a better isolation of the reason for the stall.

%RESOURCE.STALLS.COST = 100 * RESOURCE_STALLS.ANY / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.SB - Occurs when a store micro-op is ready for allocation and all store buffer entries are in use, usually due to long latency stores in progress. Typically this event tags to the IP after the store instruction that is stalled at allocation.

%RESOURCE.STALLS.SB.COST = 100 * RESOURCE_STALLS.SB / CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.LB - Counts cycles in which a load micro-op is ready for allocation and all load buffer entries are taken, usually due to long latency loads in progress. In many cases the queue to the scheduler becomes full by micro-ops that depend on the long latency loads, before the load buffer gets full.

%RESOURCE.STALLS.LB.COST = 100 * RESOURCE_STALLS.LB / CPU_CLK_UNHALTED.THREAD;

In the above cases the event RESOURCE_STALLS.RS will often count in parallel. The best methodology to further investigate loss in data locality is the high cache line replacement study described in Section 22.5.4.2, concentrating on L1 DCache replacements first

RESOURCE_STALLS.RS - Scheduler slots are typically the first resource that runs out when the pipeline is backed up. However, this can be due to almost any bottleneck in the back end, including long latency loads and instructions backed up at the execute stage. Thus it is recommended to investigate other resource stalls, before digging into the stalls tagged to lack of scheduler entries. The skid tends to be low on this event.

%RESOURCE.STALLS.RS.COST = 100 * RESOURCE_STALLS.RS/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS.ROB - Counts cycles when allocation stalls because all the reorder buffer (ROB) entries are taken. This event occurs less frequently than the RESOURCE_STALLS.RS and typically indicates that the pipeline is being backed up by a micro-op that is holding all younger micro-ops from retiring because they have to retire in order.

%RESOURCE.STALLS.ROB.COST = 100 * RESOURCE_STALLS.ROB/ CPU_CLK_UNHALTED.THREAD;

RESOURCE_STALLS2.BOB_FULL - Counts when allocation is stalled due to a branch micro-op that is ready for allocation, but the number of branches in progress in the processor has reached the limit.

%RESOURCE.STALLS.BOB.COST = 100 * RESOURCE_STALLS2.BOB/ CPU_CLK_UNHALTED.THREAD;

22.5.4 MEMORY SUB-SYSTEM STALLS

The following subsections discusses using specific performance monitoring events in Sandy Bridge microarchitecture to identify stalls in the memory sub-systems.

22.5.4.1 Accounting for Load Latency

The breakdown of load operation locality can be accomplished at any granularity including process, module, function and instruction. When you find that a load instruction is a bottleneck, investigate it further with the precise load breakdown. If this does not explain the bottleneck, check for other issues which can impact loads.

You can use these events to estimate the costs of the load causing a bottleneck, and to obtain a percentage breakdown of memory hierarchy level. Not all tools provide support for precise event sampling. If the precise version (event name ends with a suffix PS) of these event is not supported in a given tool, you can use the non-precise version.

The precise load events tag the event to the next instruction retired (IP+1).

Required events

MEM_LOAD_UOPS_RETIRED.L1_HIT_PS - Counts demand loads that hit the first level of the data cache, the L1 DCache. Demand loads are non speculative load micro-ops.

MEM_LOAD_UOPS_RETIRED.L2_HIT_PS - Counts demand loads that hit the 2nd level cache, the L2.

MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS - Counts demand loads that hit the 3rd level shared cache, the LLC.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS - Counts demand loads that hit the 3rd level shared cache and are assumed to be present also in a cache of another core but the cache line was already evicted from there.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS - Counts demand loads that hit a cache line in a cache of another core and the cache line has not been modified.

MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS - Counts demand loads that hit a cache line in the cache of another core and the cache line has been written to by that other core. This event is important for many performance bottlenecks that can occur in multi-threaded applications, such as lock contention and false sharing.

MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS - Counts demand loads that missed the LLC. This means that the load is usually satisfied from memory in client system.

MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS - Counts demand loads that hit in the line fill buffer (LFB). A LFB entry is allocated every time a miss occurs in the L1 DCache. When a load hits at this location it means that a previous load, store or hardware prefetch has already missed in the L1 DCache and the data fetch is in progress. Therefore the cost of a hit in the LFB varies. This event may count cache-line split loads that miss in the L1 DCache but do not miss the LLC.

On 32-byte Intel AVX loads, all loads that miss in the L1 DCache show up as hits in the L1 DCache or hits in the LFB. They never show hits on any other level of memory hierarchy. Most loads arise from the line fill buffer (LFB) when Intel AVX loads miss in the L1 DCache.

Precise Load Breakdown

The percentage breakdown of each load source can be tagged at any granularity including a single IP, function, module, or process. This is particularly useful at a single instruction to determine the breakdown of where the load was found in the cache hierarchy. The following formula shows how to calculate the percentage of time a load was satisfied by the LLC. Similar formulas can be used for all other hierarchy levels.

%LocL3.HIT =

100 * MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS / $SumOf_PRECISE_LOADS;

$SumOf_PRECISE_LOADS =

MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS +MEM_LOAD_UOPS_RETIRED.L1_HIT_PS +
MEM_LOAD_UOPS_RETIRED.L2_HIT_PS +MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS +
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS +
MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS;

Estimated Load Penalty

The formulas below help estimating to what degree loads from a certain memory hierarchy are responsible for a slowdown. The CPU_CLK_UNHALTED.THREAD programmable event represents the penalty in cycles tagged at the same granularity. At the instruction level, the cycle cost of an expensive load tends to only skid one IP, similar to the precise event. The calculations below apply to any granularity process, module, function or instruction, since the events are precise. Anything representing 10%, or higher, of the total clocks in a granularity of interest should be investigated.

If the code has highly dependent loads you can use the MEM_LOAD_UOPS_RETIRED.L1_HIT_PS event to determine if the loads are hit by the five cycle latency of the L1 DCache.

Estimated cost of L2 latency

%L2.COST =

12 * MEM_LOAD_UOPS_RETIRED.L2_HIT_PS / CPU_CLK_UNHALTED.THREAD;

Estimated cost of L3 hits

%L3.COST =

26 * MEM_LOAD_UOPS_RETIRED.L3_HIT_PS / CPU_CLK_UNHALTED.THREAD;

Estimated cost of hits in the cache of other cores

%HIT.COST =

43* MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS /
CPU_CLK_UNHALTED.THREAD;

Estimated cost of memory latency

%MEMORY.COST =

200 * MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS_PS /
CPU_CLK_UNHALTED.THREAD;

Actual memory latency can vary greatly depending on memory parameters. The amount of concurrent memory traffic often reduces the effect cost of a given memory hierarchy. Typically, the estimates above may be on the pessimistic side (like pointer-chasing situations).

Often, cache misses will manifest as delaying and bunching on the retirement of instructions. The precise loads breakdown can provide estimates of the distribution of hierarchy levels where the load is satisfied.

Given a significant impact from a particular cache level, the first step is to find where heavy cache line replacements are occurring in the code. This could coincide with your hot portions of code detected by the memory hierarchy breakdown, but often does not. For instance, regular traversal of a large data structure can unintentionally clear out levels of cache.

If hits of non modified or modified data in another core have high estimated cost and are hot at locations in the code, it can be due to locking, sharing or false sharing issues between threads.

If load latency in memory hierarchy levels further from the L1 DCache does not justify the amount of cycles spent on a load, try one of the following:

- Eliminate superfluous load operations such as spilling general purpose registers to XMM registers rather than memory.
- Continue searching for issues impacting load instructions described in Section 22.5.4.4.

22.5.4.2 Cache-line Replacement Analysis

When an application has many cache misses, it is a good idea to determine where cache lines are being replaced at the highest frequency. The instructions responsible for high amount of cache replacements are not always where the application is spending most of its time, since replacements can be driven by the hardware prefetchers and store operations which in the common case do not hold up the pipeline. Typically traversing large arrays or data structures can cause heavy cache line replacements.

Required events:

- L1D.REPLACEMENT - Replacements in the 1st level data cache.
- L2_LINES_IN.ALL - Cache lines being brought into the L2 cache.

Usages of events:

Identifying the replacements that potentially cause performance loss can be done at process, module, and function level. Do it in two steps:

- Use the precise load breakdown to identify the memory hierarchy level at which loads are satisfied and cause the highest penalty.
- Identify, using the formulas below, which portion of code causes most of the replacements in the level below the one that satisfies these high penalty loads.

For example, if there is high penalty due to loads hitting the LLC, check the code which is causing replacements in the L2 and the L1. In the formulas below, the nominators are the replacements accounted for a module or function. The sum of the replacements in the denominators is the sum of all replacements in a cache level for all processes. This enables you to identify the portion of code that causes most of the replacements.

L1D Cache Replacements

%L1D.REPLACEMENT =

L1D.REPLACEMENT / SumOverAllProcesses(L1D.REPLACEMENT);

L2 Cache Replacements

%L2.REPLACEMENT =

L2_LINES_IN.ALL / SumOverAllProcesses(L2_LINES_IN.ALL);

22.5.4.3 Lock Contention Analysis

The amount of contention on locks is critical in scalability analysis of multi-threaded applications. A typical ring3 lock almost always results in the execution of an atomic instruction. An atomic instruction is either an XCHG instruction

involving a memory address or one of the following instructions with memory destination and lock prefix: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR or XADD. Precise events enable you to get an idea of the contention on any lock. Many locking APIs start by an atomic instruction in ring3 and back off a contended lock by jumping into ring0. This means many locking APIs can be very costly in low contention scenarios. To estimate the amount of contention on a locked instruction, you can measure the number of times the cache line containing the memory destination of an atomic instruction is found modified in another core.

Required events:

- MEM_UOPS_RETIRED.LOCK_LOADS_PS - Counts the number of atomic instructions which are retired with a precise skid of IP+1.
- MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS - Counts the occurrences that the load hits a modified cache line in another core. This event is important for many performance bottlenecks that can occur in multi-core systems, such as lock contention, and false sharing.

Usages of events:

The lock contention factor gives the percentage of locked operations executed that contend with another core and therefore have a high penalty. Usually a lock contention factor over 5% is worth investigating on a hot lock. A heavily contended lock may impact the performance of multiple threads.

```
%LOCK.CONTENTION =
        100 * MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS /
        MEM_UOPS_RETIRED.LOCK_LOAD_PS;
```

22.5.4.4 Other Memory Access Issues

Store Forwarding Blocked

When store forwarding is not possible the dependent loads are blocked. The average penalty for store forward block is 13 cycles. Since many cases of store forwarding blocked were fixed in prior architectures, the most common case in code today involves storing to a smaller memory space than an ensuing larger load.

Required events:

LD_BLOCKS.STORE_FORWARD - Counts the number of times a store forward opportunity is blocked due to the inability of the architecture to forward a small store to a larger load and some rare alignment cases.

Usages of Events:

Use the following formula to estimate the cost of the store forward block. The event LD_BLOCKS.STORE_FORWARD tends to be tagged to the next IP after the attempted load, so it is recommended to look at this issue at the instruction level. However it is possible to inspect the ratio at any granularity: process, module, function or IP.

```
%STORE.FORWARD.BLOCK.COST =
        100 *LD_BLOCKS.STORE_FORWARD * 13 / CPU_CLK_UNHALTED.THREAD;
```

After finding a load blocked from store-forwarding, the location of the store must also be found. Typically, about 60% of all store forwarded blocked issue are caused by stores in the last 10 instructions executed prior to the load.

The most common case in which store forward blocked is seen is a small store that is unable to forward to a larger load. For example, the code below generated writes to a byte pointer address and then reads from a four byte (dword) memory space:

```
and     byte ptr [ebx],7f
and     dword ptr [ebx], ecx
```

To fix a store forward block, it's best to fix the store operation and not the load.

Cache Line Splits

Beginning with Nehalem microarchitecture, the L1 DCache has split registers which enable it to handle loads and stores that span two cache lines in a faster manner. This puts the cost of split loads at about five cycles, as long as split registers are available, instead of the 20 cycles required in earlier microarchitectures. Handling of split stores handling is usually hidden, but if there are many of them they can stall allocation due to a full store buffer, or they can consume split registers that may be needed for handling split loads. Getting quantifiable gains from eliminating cache line splits is still achievable.

Required events:

- MEM_UOPS_RETIRED.SPLIT_LOADS_PS - Counts the number of demand loads that span two cache lines. The event is precise.
- MEM_UOPS_RETIRED.SPLIT_STORES_PS - Counts the number of stores that span two cache lines. The event is precise.

Usages of events:

Finding split loads is fairly easy because they usually tag most of their cost to the next IP which is executed. The ratio below can be used at any granularity: process, module, function, and IP after split.

%SPLIT.LOAD.COST =

100 * MEM_UOPS_RETIRED.SPLIT_STORES_PS * 5 / CPU_CLK_UNHALTED.THREAD;

Split store penalty is more difficult to find using an estimated cost, because in typical cases stores do not push out the retirement of instructions. To detect significant amount of split stores divide their number by the total number of stores retired at that IP.

SPLIT.STORE.RATIO =

MEM_UOPS_RETIRED.SPLIT_STORES_PS / MEM_UOPS_RETIRED.ANY_STORES_PS;

4k Aliasing

A 4k aliasing conflict between loads and stores causes a reissue on the load. Five cycles is used as an estimate in the model below.

Required Events:

- LD_BLOCKS_PARTIAL.ADDRESS_ALIAS - Counts the number of loads that have partial address match with preceding stores, causing the load to be reissued.

Usages of events:

%4KALIAS.COST =

100 * LD_BLOCK_PARTIAL.ADDRESS_ALIAS * 5 / CPU_CLK_UNHALTED.THREAD;

Load and Store Address Translation

There are two levels of translation look-aside buffer (TLB) for linear to physical address translation. A miss in the DTLB, the first level TLB, that hits in the STLB, the second level TLB, incurs a seven cycle penalty.

Missing in the STLB requires the processor to walk through page table entries that contain the address translation. These walks have variable cost depending on the location of the page table entries. The walk duration is a fairly good estimate of the cost of STLB misses.

Required events:

- DTLB_LOAD_MISSES.STLB_HIT - Counts loads that miss the DTLB and hit in the STLB. This event has a low skid and hence can be used at the IP level.
- DTLB_LOAD_MISSES.WALK_DURATION - Duration of a page walks in cycles following STLB misses. Event skid is typically one instruction, enabling you to detect the issue at instruction, function, module or process granularities.
- MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS - Precise event for loads which have their translation miss the STLB. The event counts only the first load from a page that initiates the page walk.
- Usage of events:

Cost of STLB hits on loads:

%STLB.HIT.COST =

100 * DTLB_LOAD_MISSES.STLB_HIT * 7/ CPU_CLK_UNHALTED.THREAD;

- Cost of page walks:

%STLB.LOAD.MISS.WALK.COST =

100 * DTLB_LOAD_MISSES.WALK_DURATION / CPU_CLK_UNHALTED.THREAD;

Use the precise STLB miss event at the IP level to determine exactly which instruction and source line suffers from frequent STLB misses.

%STLB.LOAD.MISS =

100 * MEM_UOPS_RETIRED.STLB_MISS_LOADS_PS/

MEM_UOPS_RETIRED.ANY_LOADS_PS;

Large walk durations, of hundreds of cycles, are an indication that the page tables have been thrown out of the LLC. To determine the average cost of a page walk use the following ratio:

STLB.LOAD.MISS.AVGCOST =

DTLB_LOAD_MISSES.WALK_DURATION /

DTLB_LOAD_MISSES.WALK_COMPLETED;

To a lesser extent than loads, STLB misses on stores can be a bottleneck. If the store itself is a large bottleneck, cycles will tag to the next IP after the store.

%STLB.STORE.MISS =

100 * MEM_UOPS_RETIRED.STLB_MISS_STORES_PS/

MEM_UOPS_RETIRED.ANY_STORES_PS;

Reducing DTLB/STLB misses increases data locality. One may consider using an commercial-grade memory allocators to improve data locality. Compilers which offer profile guided optimizations may reorder global variables to increase data locality, if the compiler can operate on the whole module. For issues with a large amount of time spent in page walks, server and HPC applications may be able to use large pages for data.

22.5.5 EXECUTION STALLS

The following subsections discuss using specific performance monitoring events in Sandy Bridge microarchitecture to identify stalls in the out-of-order engine.

22.5.5.1 Longer Instruction Latencies

Some microarchitectural changes manifested in longer latency for some legacy instructions in existing code. It is possible to detect some of these situations:

- Three-operand slow LEA instructions (see Section 3.5.2).
- Flags merge micro-op - These merges are primarily required by “shift cl” instructions (see Section 3.5.2.5).

These events tend to have a skid as high as a ten instructions because the eventing condition is detected early in the pipeline.

Event usage:

To use this event effectively without being distracted by the event skid, you can use it to locate performance issue at the process, module and function granularities, but not at the instruction level. To identify issue at the instruction IP granularity, one can perform static analysis on the functions identified by this event. To estimate the contribution of these events to the code latency, divide them by the cycles at the same granularity. To estimate the overall impact, start with the total cycles due to these issues and if significant continue to search for the exact reason using the sub events.

Total cycles spent in the specified scenarios:

Flags Merge micro-op ratio:

%FLAGS.MERGE.UOP =

100 * PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP_CYCLES /

CPU_CLK_UNHALTED.THREAD;

Slow LEA instructions allocated:

%SLOW.LEA.WINDOW =

100 * PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW / CPU_CLK_UNHALTED.THREAD;

22.5.5.2 Assists

Assists usually involve the microcode sequencer that helps handle the assist. Determining the number of cycles where microcode is generated from the microcode sequencer is often a good methodology to determine the total cost of the assist. If the overall cost of assists are high, a breakdown of assists into specific types will be useful.

- Estimating the total cost of assists using microcode sequencer cycles:

%ASSISTS.COST =

100 * IDQ.MS_CYCLES / CPU_CLK_UNHALTED.THREAD;

Floating-point assists

Denormal inputs for X87 instructions require an FP assist, potentially costing hundreds of cycles.

%FP.ASSISTS =

100 *FP_ASSIST.ANY / INST_RETIRED.ANY;

Transitions between Intel SSE and Intel AVX

The transitions between Intel SSE and Intel AVX code are explained in detail in Section 15.3.1. The typical cost is about seventy-five cycles.

%AVX2SSE.TRANSITION.COST =

75 * OTHER_ASSISTS.AVX_TO_SSE / CPU_CLK_UNHALTED.THREAD;

%SSE2AVX.TRANSITION.COST =

75 * OTHER_ASSISTS.SSE_TO_AVX / CPU_CLK_UNHALTED.THREAD;

32-byte AVX store instructions that span two pages require an assist that costs roughly 150 cycles. A large amount of microcode tagged to the IP after a 32-byte AVX store is a good sign that an assist has occurred.

%AVX.STORE.ASSIST.COST =

150 * OTHER_ASSISTS.AVX_STORE / CPU_CLK_UNHALTED.THREAD;

22.5.6 BAD SPECULATION

This section discusses mispredicted branch instructions resulting in a pipeline flush.

22.5.6.1 Branch Mispredicts

The largest challenge with mispredicted branches is finding the branch which caused them. Branch mispredictions incur penalty of about 20 cycles. The cost varies based upon the misprediction, and whether the correct path is found in the Decoded ICache or in the legacy decode pipeline.

Required Events:

BR_MISP_RETIRED.ALL_BRANCHES_PS is a precise event that counts branches that incorrectly predicted the branch target. Since this is a precise event that skids to the next instruction, it tags to the first instruction in the correct path

after the branch misprediction. This study can be performed at the process, module, function or instruction granularity.

Usages of Events:

Use the following ratio to estimate the cost of mispredicted branches:

%BR.MISP.COST =

20 * BR_MISP_RETIRED.ALL_BRANCHES_PS / CPU_CLK_UNHALTED.THREAD;

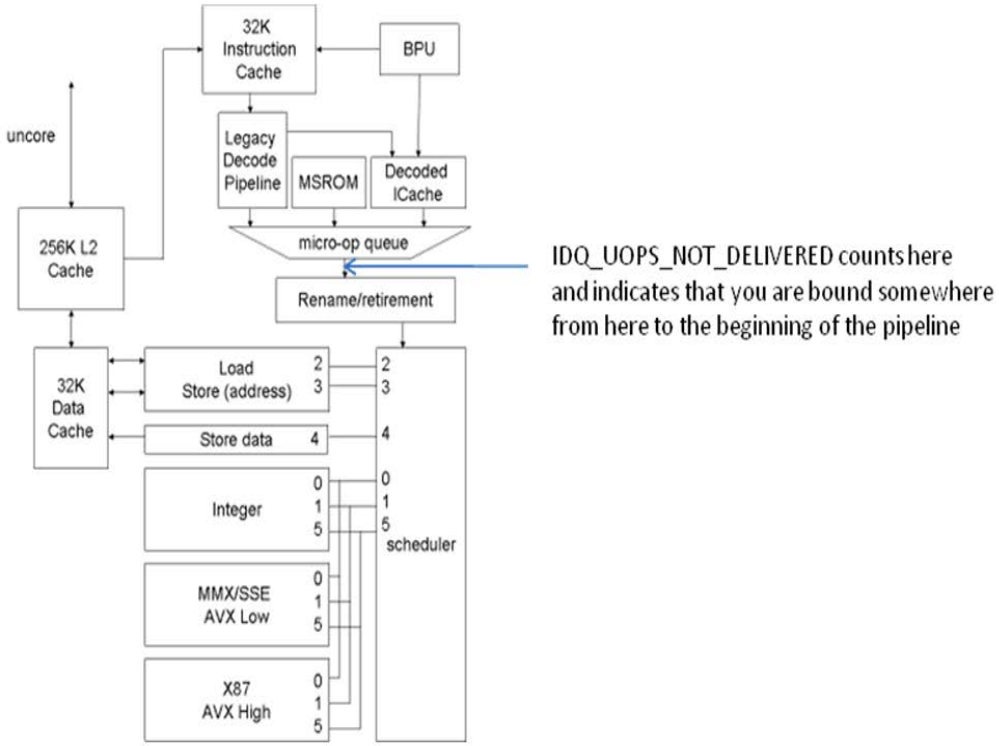
22.5.7 FRONTEND STALLS

Stalls in the Frontend should not be investigated unless the analysis in Section 22.5.2 showed at least 30% of a granularity being bound in the front end. This section explains the main issues that can cause delays in the front end of the pipeline. Events detected in the front end have unpredictable skid. Therefore do not try and associate the penalty at the IP level. Stay at the function, module, and process level for these events.

22.5.7.1 Understanding the Micro-op Delivery Rate

Usages of Counters

The event IDQ_UOPS_NOT_DELIVERED counts when the maximum of four micro-ops are not delivered to the rename stage, while it is requesting micro-ops. When the pipeline is backed up the rename stage does not request any further micro-ops from the front end. The diagram above shows how this event tracks micro-ops between the micro-op queue and the rename stage.



You can use the IDQ_UOPS_NOT_DELIVERED event to breakdown the distribution of cycles when 0, 1, 2, 3 micro-ops are delivered from the front end.

Percentage of cycles the front end is effective, or execution is back end bound:

%FE.DELIVERING =

100 * (CPU_CLK_UNHALTED.THREAD -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering three micro-ops per cycle:

%FE.DELIVER.3UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering two micro-ops per cycle:

%FE.DELIVER.2UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering one micro-ops per cycle:

%FE.DELIVER.1UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE -
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Percentage of cycles the front end is delivering zero micro-ops per cycle:

%FE.DELIVER.0UOPS =

100 * (IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE) /
CPU_CLK_UNHALTED.THREAD;

Average Micro-ops Delivered per Cycle: This ratio assumes that the front end could potentially deliver four micro-ops per cycle when bound in the back end.

AVG.uops.per.cycle =

(4 * (%FE.DELIVERING) + 3 * (%FE.DELIVER.3UOPS) + 2 * (%FE.DELIVER.2UOPS) +
(%FE.DELIVER.1UOPS)) / 100

Seeing the distribution of the micro-ops being delivered in a cycle is a hint at the front end bottlenecks that might be occurring. Issues such as LCPs and penalties from switching from the decoded ICache to the legacy decode pipeline tend to result in zero micro-ops being delivered for several cycles. Fetch bandwidth issues and decoder stalls result in less than four micro-ops delivered per cycle.

22.5.7.2 Understanding the Sources of the Micro-op Queue

The micro-op queue can get micro-ops from the following sources:

- Decoded ICache.
- Legacy decode pipeline.
- Microcode sequencer (MS).

A typical distribution is approximately 80% of the micro-ops coming from the Decoded ICache, 15% coming from legacy decode pipeline and 5% coming from the microcode sequencer. Excessive micro-ops coming from the legacy decode pipeline can be a warning sign that the Decoded ICache is not working effectively. A large portion of micro-ops coming from the microcode sequencer may be benign, such as complex instructions, or string operations, but can also be due to code assists handling undesired situations like Intel SSE to Intel AVX code transitions.

Description of Counters Required

- IDQ.DSB_UOPS - Micro-ops delivered to the micro-op queue from the Decoded ICache.
- IDQ.MITE_UOPS - Micro-ops delivered to the micro-op queue from the legacy decode pipeline.
- IDQ.MS_UOPS -Micro-ops delivered from the microcode sequencer.

Usage of Counters

Percentage of micro-ops coming from Decoded ICache

%UOPS.DSB =

IDQ.DSB_UOPS / ALL_IDQ_UOPS;

Percentage of micro-ops coming from legacy decoder pipeline:

%UOPS.MITE =

IDQ.MITE_UOPS / ALL_IDQ_UOPS;

Percentage of micro-ops coming from micro-sequencer:

%UOPS.MS =

IDQ.MS_UOPS / ALL_IDQ_UOPS;

ALL_IDQ_UOPS = (IDQ.DSB_UOPS + IDQ.MITE_UOPS + IDQ.MS_UOPS);

If your application is not bound in the front end then whether micro-ops are coming from the legacy decode pipeline or Decoded ICache is of lesser importance. Excessive micro-ops coming from the microcode sequencer are worth investigating further to see if assists might be a problem.

Cases to investigate are listed below:

- **(**%FE_BOUND > 30%) and (%UOPS.DSB < 70%)

  A threshold of 30% defines a "front end bound" case. This threshold may be applicable to many situations, but may also vary somewhat across different workloads.

  — Investigate why micro-ops are not coming from the Decoded ICache.

  — Investigate issues which can impact the legacy decode pipeline.
- (%FE_BOUND > 30%) and (%UOP_DSB > 70%)

  — Investigate switches from Decoded ICache to legacy decode pipeline since it may be switching to run portions of code that are too small to be effective.

  — Look at the amount of bad speculation, since branch mispredictions still impact FE performance.

  — Determine the average number of micro-ops being delivered per 32-byte chunk hit. If there are many taken branches from one 32-byte chunk into another, it impacts the micro-ops being delivered per cycle.

  — Micro-op delivery from the Decoded ICache may be an issue which is not covered.
- (%FE_BOUND < 20%) and (%UOPS_MS>25%)

  A threshold of 20% defines a "front end not bound" case. This threshold may be applicable to many situations, but may also vary somewhat across different workloads.

  The following steps can help determine why micro-ops came from the microcode, in order of most common to least common.

— Long latency instructions - Any instruction over four micro-ops starts the microcode sequencer. Some instructions such as transcendentals can generate many micro-ops from the microcode.

— String operations - string operations can produce a large amount of microcode. In some cases there are assists which can occur due to string operations such as REP MOVSB with trip count greater than 3, which costs 70+ cycles.

— Assists - See Section 22.5.5.2.

22.5.7.3 The Decoded ICache

The Decoded ICache has many advantages over the legacy decode pipeline. It eliminates many bottlenecks of the legacy decode pipeline such as instructions decoded into more than one micro-op and length changing prefix (LCP) stalls.

A switch to the legacy decode pipeline from the Decoded ICache only occurs when a lookup in the Decoded ICache fails and usually costs anywhere from zero to three cycles in the front end of the pipeline.

Required Events

The Decoded ICache events all have large skids and the exact instruction where they are tagged is usually not the source of the problem so only look for this issue at the process, module and function granularities.

- DSB2MITE_SWITCHES.PENALTY_CYCLES - Counts the cycles attributed to the switch from the Decoded ICache to the legacy decode pipeline, excluding cycles when the micro-op queue cannot accept micro-ops because it is back end bound.
- DSB2MITE_SWITCHES.COUNT - Counts the number of switches between the Decoded ICache and the legacy decode pipeline.
- DSB_FILL.ALL_CANCEL - Counts when fills to the Decoded ICache are canceled.
- DSB_FILL.EXCEED_DSB_LINES- Counts when a fill is canceled because the allocated lines for Decoded ICache has exceeded three for the 32-byte chunk.

Usage of Events

Since these studies involve front end events, do not try to tag the event to a specific instruction.

- Determining cost of switches from the Decoded ICache to the legacy decode pipeline.

  %DSB2MITE.SWITCH.COST =

  100 * DSB2MITE_SWITCHES.PENALTY_CYCLES / CPU_CLK_UNHALTED.THREAD;

- Determining the average cost per Decoded ICache switch to the legacy front end:

  AVG.DSB2MITE.SWITCH.COST =

  DSB2MITE_SWITCHES.PENALTY_CYCLES / DSB2MITE_SWITCHES.COUNT;

- Determining Causes of Misses in the Decoded ICache
  - There are no partial hits in the Decoded ICache. If any micro-op that is part of that lookup on the 32-byte chunk is missing, a Decoded ICache miss occurs on all micro-ops for that transaction.
  - There are three primary reasons for missing micro-ops in the Decoded ICache:
    - **Portions of a 32-byte chunk of code were not able to fit within three ways of the Decoded ICache.**
    - **A frequently run portion of your code section is too large for the Decoded ICache. This case is more common on server applications since client applications tend to have a smaller set of code which is "hot".**
    - **The Decoded ICache is getting flushed for example when an ITLB entry is evicted.**

— To determine if a portion of the 32-byte code is unable to fit into three lines within the Decoded ICache use the DSB_FILL.EXCEED_DSB_LINESevent at the process, module or function granularities

%DSB.EXCEED.WAY.LIMIT =

100 * DSB_FILL.EXCEED_DSB_LINES/ DSB_FILL.ALL_CANCEL;

22.5.7.4 Issues in the Legacy Decode Pipeline

If a large percentage of the micro-ops going to the micro-op queue are being delivered from the legacy decode pipeline, you should check to see if there are bottlenecks impacting that stage. The most common bottlenecks in the legacy decode pipeline are:

- Fetch not providing enough instructions.

  This happens when hot code is poorly aligned. For example if the hot code being fetched to be run is on the 15th byte, then only one byte is fetched.
- Length changing prefix stalls in the instruction length decoder.

  Instructions that are decoded into two to four micro-ops may introduce a bubble in the decoder throughput. If the instruction queue, preceding the decoders, becomes full, this indicates that these instructions may cause a penalty.

%ILD.STALL.COST =

100 * ILD_STALL.LCP * 3 / CPU_CLK_UNHALTED.THREAD;

22.5.7.5 Instruction Cache

Applications with large hot code sections tend to run into many issues with the instruction cache. This is more typical in server applications.

Required Events

ICACHE.MISSES - Counts the number of instruction byte fetches that miss the ICache

Usage of Events

To determine whether ICache misses are causing the issue, compare them to the instructions retired event count, using the same granularity (process, model, or function). Anything over 1% of instructions retired can be a significant issue.

ICACHE.PER.INST.RET =

ICACHE.MISSES / INST_RETIRED.ANY;

If ICache misses are causing a significant problem, try to reduce the size of your hot code section, using the profile guided optimizations. Most compilers have options for text reordering which helps reduce the number of pages and, to a lesser extent, the number of pages your application is covering.

If the application makes significant use of macros, try to either convert them to functions, or use intelligent linking to eliminate repeated code.

22.6 USING PERFORMANCE EVENTS OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

There are performance events specific to the microarchitecture of Intel Core Solo and Intel Core Duo processors. Lists of available performance-monitoring events can be found *at:*

https://perfmon-events.intel.com/.

22.6.1 UNDERSTANDING THE RESULTS IN A PERFORMANCE COUNTER

Each performance event detects a well-defined microarchitectural condition occurring in the core while the core is active. A core is active when:

- It's running code (excluding the halt instruction).
- It's being snooped by the other core or a logical processor on the platform. This can also happen when the core is halted.

Some microarchitectural conditions are applicable to a sub-system shared by more than one core and some performance events provide an event mask (or unit mask) that allows qualification at the physical processor boundary or at bus agent boundary.

Some events allow qualifications that permit the counting of microarchitectural conditions associated with a particular core versus counts from all cores in a physical processor (see L2 and bus related events *at: https://perfmon-events.intel.com/*).

When a multi-threaded workload does not use all cores continuously, a performance counter counting a core-specific condition may progress to some extent on the halted core and stop progressing or a unit mask may be qualified to continue counting occurrences of the condition attributed to either processor core. Typically, one can adjust the highest two bits (bits 15:14 of the IA32_PERFEVTSELx MSR) in the unit mask field to distinguish such asymmetry.

See Chapter 18, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

There are three cycle-counting events which will not progress on a halted core, even if the halted core is being snooped. These are: Unhalted core cycles, Unhalted reference cycles, and Unhalted bus cycles. All three events are detected for the unit selected by event 3CH.

Some events detect microarchitectural conditions but are limited in their ability to identify the originating core or physical processor. For example, bus_drdy_clocks may be programmed with a unit mask of 20H to include all agents on a bus. In this case, the performance counter in each core will report nearly identical values. Performance tools interpreting counts must take into account that it is only necessary to equate bus activity with the event count from one core (and not use not the sum from each core).

The above is also applicable when the core-specificity sub field (bits 15:14 of IA32_PERFEVTSELx MSR) within an event mask is programmed with 11B. The result of reported by performance counter on each core will be nearly identical.

22.6.2 RATIO INTERPRETATION

Ratios of two events are useful for analyzing various characteristics of a workload. It may be possible to acquire such ratios at multiple granularities, for example: (1) per-application thread, (2) per logical processor, (3) per core, and (4) per physical processor.

The first ratio is most useful from a software development perspective, but requires multi-threaded applications to manage processor affinity explicitly for each application thread. The other options provide insights on hardware utilization.

In general, collect measurements (for all events in a ratio) in the same run. This should be done because:

- If measuring ratios for a multi-threaded workload, getting results for all events in the same run enables you to understand which event counter values belongs to each thread.
- Some events, such as writebacks, may have non-deterministic behavior for different runs. In such a case, only measurements collected in the same run yield meaningful ratio values.

22.6.3 NOTES ON SELECTED EVENTS

This section provides event-specific notes for interpreting performance events listed *at: https://perfmon-events.intel.com/*.

- **L2_Reject_Cycles, event number 30H** — This event counts the cycles during which the L2 cache rejected new access requests.
- **L2_No_Request_Cycles, event number 32H** — This event counts cycles during which no requests from the L1 or prefetches to the L2 cache were issued.
- **Unhalted_Core_Cycles, event number 3C, unit mask 00H** — This event counts the smallest unit of time recognized by an active core.

  In many operating systems, the idle task is implemented using HLT instruction. In such operating systems, clock ticks for the idle task are not counted. A transition due to Enhanced Intel SpeedStep Technology may change the operating frequency of a core. Therefore, using this event to initiate time-based sampling can create artifacts.
- **Unhalted_Ref_Cycles, event number 3C, unit mask 01H** — This event guarantees a uniform interval for each cycle being counted. Specifically, counts increment at bus clock cycles while the core is active. The cycles can be converted to core clock domain by multiplying the bus ratio which sets the core clock frequency.
- **Serial_Execution_Cycles, event number 3C, unit mask 02H** — This event counts the bus cycles during which the core is actively executing code (non-halted) while the other core in the physical processor is halted.
- **L1_Pref_Req, event number 4FH, unit mask 00H** — This event counts the number of times the Data Cache Unit (DCU) requests to prefetch a data cache line from the L2 cache. Requests can be rejected when the L2 cache is busy. Rejected requests are re-submitted.
- **DCU_Snoop_to_Share, event number 78H, unit mask 01H** — This event counts the number of times the DCU is snooped for a cache line needed by the other core. The cache line is missing in the L1 instruction cache or data cache of the other core; or it is set for read-only, when the other core wants to write to it. These snoops are done through the DCU store port. Frequent DCU snoops may conflict with stores to the DCU, and this may increase store latency and impact performance.
- **Bus_Not_In_Use, event number 7DH, unit mask 00H** — This event counts the number of bus cycles for which the core does not have a transaction waiting for completion on the bus.
- **Bus_Snoops, event number 77H, unit mask 00H** — This event counts the number of CLEAN, HIT, or HITM responses to external snoops detected on the bus.

  In a single-processor system, CLEAN and HIT responses are not likely to happen. In a multiprocessor system this event indicates an L2 miss in one processor that did not find the missed data on other processors.

  In a single-processor system, an HITM response indicates that an L1 miss (instruction or data) found the missed cache line in the other core in a modified state. In a multiprocessor system, this event also indicates that an L1 miss (instruction or data) found the missed cache line in another core in a modified state.

22.7 DRILL-DOWN TECHNIQUES FOR PERFORMANCE ANALYSIS

Software performance intertwines code and microarchitectural characteristics of the processor. Performance monitoring events provide insights to these interactions. Each microarchitecture often provides a large set of performance events that target different sub-systems within the microarchitecture. Having a methodical approach to select key performance events will likely improve a programmer's understanding of the performance bottlenecks and improve the efficiency of code-tuning effort.

Recent generations of Intel 64 and IA-32 processors feature microarchitectures using an out-of-order execution engine. They are also accompanied by an in-order front end and retirement logic that enforces program order. Superscalar hardware, buffering and speculative execution often complicates the interpretation of performance events and software-visible performance bottlenecks.

This section discusses a methodology of using performance events to drill down on likely areas of performance bottleneck. By narrowed down to a small set of performance events, the programmer can take advantage of Intel VTune Performance Analyzer to correlate performance bottlenecks with source code locations and apply coding recommendations discussed in Chapter 3 through Chapter 11. Although the general principles of our method can be

applied to different microarchitectures, this section will use performance events available in processors based on Intel Core microarchitecture for simplicity.

Performance tuning usually centers around reducing the time it takes to complete a well-defined workload. Performance events can be used to measure the elapsed time between the start and end of a workload. Thus, reducing elapsed time of completing a workload is equivalent to reducing measured processor cycles.

The drill-down methodology can be summarized as four phases of performance event measurements to help characterize interactions of the code with key pipe stages or sub-systems of the microarchitecture. The relation of the performance event drill-down methodology to the software tuning feedback loop is illustrated in Figure 22-16.

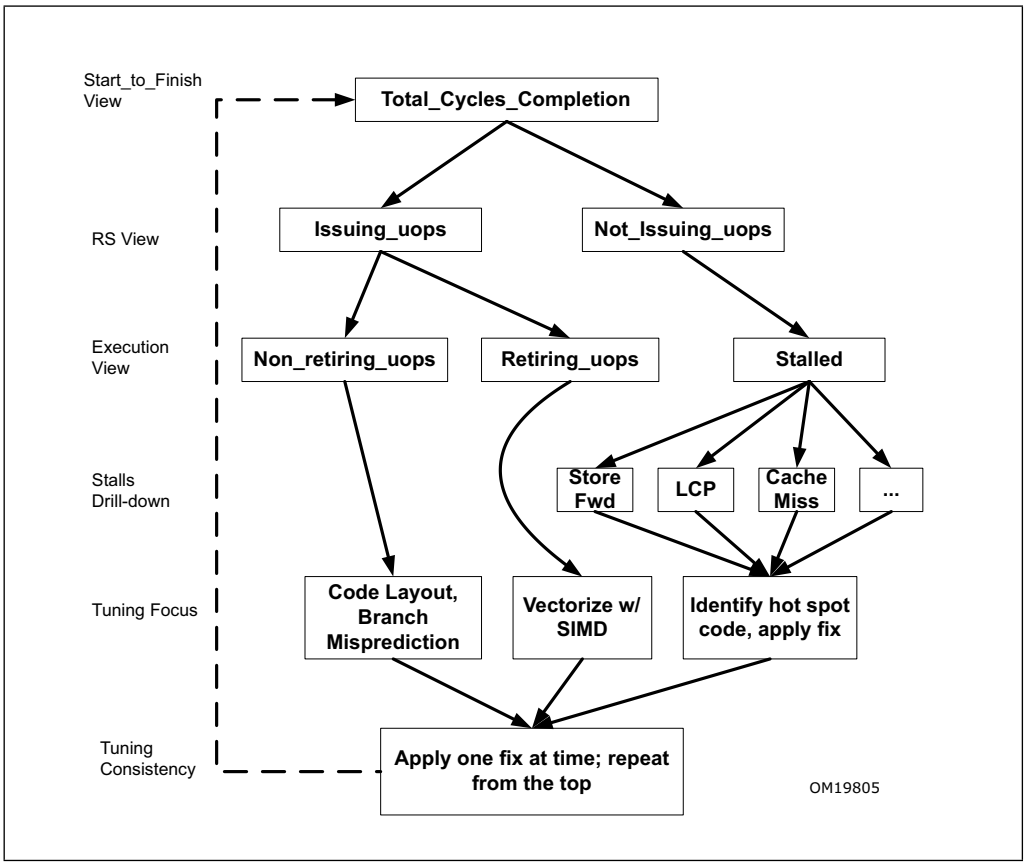


Figure 22-16. Performance Events Drill-Down and Software Tuning Feedback Loop

Typically, the logic in performance monitoring hardware measures microarchitectural conditions that varies across different counting domains, ranging from cycles, micro-ops, address references, instances, etc. The drill-down methodology attempts to provide an intuitive, cycle-based view across different phases by making suitable approximations that are described below:

- **Total cycle measurement** — This is the start to finish view of total number of cycle to complete the workload of interest. In typical performance tuning situations, the metric Total_cycles can be measured by the event CPU_CLK_UNHALTED.CORE. See*: https://perfmon-events.intel.com/*).
- **Cycle composition at issue port** — The reservation station (RS) dispatches micro-ops for execution so that the program can make forward progress. Hence the metric Total_cycles can be decomposed as consisting of two exclusive components: Cycles_not_issuing_uops representing cycles that the RS is not issuing micro-ops for execution, and Cycles_issuing_uops cycles that the RS is issuing micro-ops for execution. The latter component includes micro-ops in the architected code path or in the speculative code path.
- **Cycle composition of OOO execution** — The out-of-order engine provides multiple execution units that can execute micro-ops in parallel. If one execution unit stalls, it does not necessarily imply the program execution

is stalled. Our methodology attempts to construct a cycle-composition view that approximates the progress of program execution. The three relevant metrics are: Cycles_stalled, Cycles_not_retiring_uops, and Cycles_retiring_uops.

- **Execution stall analysis** — From the cycle compositions of overall program execution, the programmer can narrow down the selection of performance events to further pin-point unproductive interaction between the workload and a micro-architectural sub-system.

When cycles lost to a stalled microarchitectural sub-system, or to unproductive speculative execution are identified, the programmer can use VTune Analyzer to correlate each significant performance impact to source code location. If the performance impact of stalls or misprediction is insignificant, VTune can also identify the source locations of hot functions, so the programmer can evaluate the benefits of vectorization on those hot functions.

22.7.1 CYCLE COMPOSITION AT ISSUE PORT

Recent processor microarchitectures employ out-of-order engines that execute streams of micro-ops natively, while decoding program instructions into micro-ops in its front end. The metric Total_cycles alone, is opaque with respect to decomposing cycles that are productive or non-productive for program execution. To establish a consistent cycle-based decomposition, construct two metrics that can be measured using performance events available in processors based on Intel Core microarchitecture. These are:

- **Cycles_not_issuing_uops** — This can be measured by the event RS_UOPS_DISPATCHED, setting the INV bit and specifying a counter mask (CMASK) value of 1 in the target performance event select (IA32_PERFEVTSELx) MSR (See Chapter 19, “Architectural Last Branch Records” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B). In VTune Analyzer, the special values for CMASK and INV is already configured for the VTune event name RS_UOPS_DISPATCHED.CYCLES_NONE.
- **Cycles_issuing_uops** — This can be measured using the event RS_UOPS_DISPATCHED, clear the INV bit and specifying a counter mask (CMASK) value of 1 in the target performance event select MSR

Note the cycle decomposition view here is approximate in nature; it does not distinguish specificities, such as whether the RS is full or empty, transient situations of RS being empty but some in-flight µops is getting retired.

22.7.2 CYCLE COMPOSITION OF OOO EXECUTION

In an OOO engine, part of making forward progress of the program. But speculative execution of micro-ops in the shadow of mispredicted code path represent unproductive work that consumes execution resources and execution bandwidth.

Cycles_not_issuing_uops, by definition, represents the cycles that the OOO engine is stalled (Cycles_stalled). As an approximation, this can be interpreted as the cycles that the program is not making forward progress.

The micro-ops that are issued for execution do not necessarily end in retirement. Those micro-ops that do not reach retirement do not help forward progress of program execution. Hence, a further approximation is made in the formalism of decomposition of Cycles_issuing_uops into:

- **Cycles_non_retiring_uops** — Although there isn’t a direct event to measure the cycles associated with non-retiring micro-ops, derive this metric from available performance events, and several assumptions:
  - A constant issue rate of micro-ops flowing through the issue port. Thus, define: uops_rate” = “Dispatch_uops/Cycles_issuing_uops, where Dispatch_uops can be measured with RS_UOPS_DISPATCHED, clearing the INV bit and the CMASK.
  - Approximate the number of non-productive, non-retiring micro-ops by [non_productive_uops = Dispatch_uops - executed_retired_uops], where executed_retired_uops represent productive micro-ops contributing towards forward progress that consumed execution bandwidth.
  - The executed_retired_uops can be approximated by the sum of two contributions: num_retired_uops (measured by the event UOPS_RETIRED.ANY) and num_fused_uops (measured by the event UOPS_RETIRED.FUSED).

Thus, Cycles_non_retiring_uops = non_productive_uops / uops_rate.

- **Cycles_retiring_uops** — This can be derived from Cycles_retiring_uops = num_retired_uops / uops_rate.

The cycle-decomposition methodology here does not distinguish situations where productive µops and non-productive micro-ops may be dispatched in the same cycle into the OOO engine. This approximation may be reasonable because heuristically high contribution of non-retiring µops likely correlates to situations of congestions in the OOO engine and subsequently cause the program to stall.

Evaluations of these three components: Cycles_non_retiring_uops, Cycles_stalled, Cycles_retiring_uops, relative to the Total_cycles, can help steer tuning effort in the following directions:

- If the contribution from Cycles_non_retiring_uops is high, focusing on code layout and reducing branch mispredictions will be important.
- If both the contributions from Cycles_non_retiring_uops and Cycles_stalled are insignificant, the focus for performance tuning should be directed to vectorization or other techniques to improve retirement throughput of hot functions.
- If the contributions from Cycles_stalled is high, additional drill-down may be necessary to locate bottlenecks that lies deeper in the microarchitecture pipeline.

22.7.3 DRILL-DOWN ON PERFORMANCE STALLS

In some situations, it may be useful to evaluate cycles lost to stalls associated with various stress points in the microarchitecture and sum up the contributions from each candidate stress points. This approach implies a very gross simplification and introduce complications that may be difficult to reconcile with the superscalar nature and buffering in an OOO engine.

Due to the variations of counting domains associated with different performance events, cycle-based estimation of performance impact at each stress point may carry different degree of errors due to over-estimation of exposures or under-estimations.

Over-estimation is likely to occur when overall performance impact for a given cause is estimated by multiplying the per-instance-cost to an event count that measures the number of occurrences of that microarchitectural condition. Consequently, the sum of multiple contributions of lost cycles due to different stress points may exceed the more accurate metric Cycles_stalled.

However an approach that sums up lost cycles associated with individual stress point may still be beneficial as an iterative indicator to measure the effectiveness of code tuning loop effort when tuning code to fix the performance impact of each stress point. The remaining of this sub-section will discuss a few common causes of performance bottlenecks that can be counted by performance events and fixed by following coding recommendations described in this manual.

The following items discuss several common stress points of the microarchitecture:

- **L2 Miss Impact** — An L2 load miss may expose the full latency of memory sub-system. The latency of accessing system memory varies with different chipset, generally on the order of more than a hundred cycles. Server chipset tend to exhibit longer latency than desktop chipsets. The number L2 cache miss references can be measured by MEM_LOAD_RETIRED.L2_LINE_MISS.

  An estimation of overall L2 miss impact by multiplying system memory latency with the number of L2 misses ignores the OOO engine's ability to handle multiple outstanding load misses. Multiplication of latency and number of L2 misses imply each L2 miss occur serially.

  To improve the accuracy of estimating L2 miss impact, an alternative technique should also be considered, using the event BUS_REQUEST_OUTSTANDING with a CMASK value of 1. This alternative technique effectively measures the cycles that the OOO engine is waiting for data from the outstanding bus read requests. It can overcome the over-estimation of multiplying memory latency with the number of L2 misses.
- **L2 Hit Impact** — Memory accesses from L2 will incur the cost of L2 latency. The number cache line references of L2 hit can be measured by the difference between two events: MEM_LOAD_RETIRED.L1D_LINE_MISS - MEM_LOAD_RETIRED.L2_LINE_MISS.

An estimation of overall L2 hit impact by multiplying the L2 hit latency with the number of L2 hit references ignores the OOO engine's ability to handle multiple outstanding load misses.

- **L1 DTLB Miss Impact** — The cost of a DTLB lookup miss is about 10 cycles. The event MEM_LOAD_RETIRED.DTLB_MISS measures the number of load micro-ops that experienced a DTLB miss.
- **LCP Impact** — The overall impact of LCP stalls can be directly measured by the event ILD_STALLS. The event ILD_STALLS measures the number of times the slow decoder was triggered, the cost of each instance is 6 cycles
- **Store forwarding stall Impact** — When a store forwarding situation does not meet address or size requirements imposed by hardware, a stall occurs. The delay varies for different store forwarding stall situations. Consequently, there are several performance events that provide fine-grain specificity to detect different store-forwarding stall conditions. These include:
  - A load blocked by preceding store to unknown address: This situation can be measure by the event Load_-Blocks.Sta. The per-instance cost is about 5 cycles.
  - Load partially overlaps with proceeding store or 4-KByte aliased address between a load and a proceeding store: these two situations can be measured by the event Load_Blocks.Overlap_store.
  - A load spanning across cache line boundary: This can be measured by Load_Blocks.Until_Retire. The per-instance cost is about 20 cycles.

22.8 EVENT RATIOS FOR INTEL CORE MICROARCHITECTURE

Appendix 22.8 provides examples of using performance events to quickly diagnose performance bottlenecks. This section provides additional information on using performance events to evaluate metrics that can help in wide range of performance analysis, workload characterization, and performance tuning. Note that many performance event names in the Intel Core microarchitecture carry the format of XXXX.YYY. This notation derives from the general convention that XXXX typically corresponds to a unique event select code in the performance event select register (IA32_PERFEVSELx), while YYY corresponds to a unique sub-event mask that uniquely defines a specific microarchitectural condition (See Chapter 19, "Architectural Last Branch Records" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B and event lists found *at: https://perfmon-events.intel.com/*).

22.8.1 CLOCKS PER INSTRUCTIONS RETIRED RATIO (CPI)

- Clocks Per Instruction Retired Ratio (CPI): CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY.
  - The Intel Core microarchitecture is capable of reaching CPI as low as 0.25 in ideal situations. But most of the code has higher CPI The greater value of CPI for a given workload indicate it has more opportunity for code tuning to improve performance. The CPI is an overall metric, it does not provide specificity of what microarchitectural sub-system may be contributing to a high CPI value.

The following subsections defines a list of event ratios that are useful to characterize interactions with the front end, execution, and memory.

22.8.1.1 Front End Ratios

- RS Full Ratio: RESOURCE_STALLS.RS_FULL / CPU_CLK_UNHALTED.CORE * 100
- ROB Full Ratio: RESOURCE_STALLS.ROB_FULL / CPU_CLK_UNHALTED.CORE * 100
- Load or Store Buffer Full Ratio: RESOURCE_STALLS.LD_ST / CPU_CLK_UNHALTED.CORE * 100
  - When there is a low value for the ROB Full Ratio, RS Full Ratio, and Load Store Buffer Full Ratio, and high CPI it is likely that the front end cannot provide instructions and micro-ops at a rate high enough to fill the buffers in the out-of-order engine, and therefore it is starved waiting for micro-ops to execute. In this case check further for other front end performance issues.

Code Locality

- Instruction Fetch Stall: CYCLES_L1I_MEM_STALLED / CPU_CLK_UNHALTED.CORE * 100
  - The Instruction Fetch Stall ratio is the percentage of cycles during which the Instruction Fetch Unit (IFU) cannot provide cache lines for decoding due to cache and Instruction TLB (ITLB) misses. A high value for this ratio indicates potential opportunities to improve performance by reducing the working set size of code pages and instructions being executed, hence improving code locality.
- ITLB Miss Rate: ITLB_MISS_RETIRED / INST_RETIRED.ANY
  - A high ITLB Miss Rate indicates that the executed code is spread over too many pages and cause many Instructions TLB misses. Retired ITLB misses cause the pipeline to naturally drain, while the miss stalls fetching of more instructions.
- L1 Instruction Cache Miss Rate: L1I_MISSES / INST_RETIRED.ANY
  - A high value for L1 Instruction Cache Miss Rate indicates that the code working set is bigger than the L1 instruction cache. Reducing the code working set may improve performance.
- L2 Instruction Cache Line Miss Rate: L2_IFETCH.SELF.I_STATE / INST_RETIRED.ANY
  - L2 Instruction Cache Line Miss Rate higher than zero indicates instruction cache line misses from the L2 cache may have a noticeable performance impact of program performance.

22.8.1.2 Branching and Front End

- BACLEAR Performance Impact: 7 * BACLEARS / CPU_CLK_UNHALTED.CORE
  - A high value for BACLEAR Performance Impact ratio usually indicates that the code has many branches so they cannot be consumed by the Branch Prediction Unit.
- Taken Branch Bubble: (BR_TKN_BUBBLE_1+BR_TKN_BUBBLE_2) / CPU_CLK_UNHALTED.CORE
  - A high value for Taken Branch Bubble ratio indicates that the code contains many taken branches coming one after the other and cause bubbles in the front end. This may affect performance only if it is not covered by execution latencies and stalls later in the pipe.

22.8.1.3 Stack Pointer Tracker

- ESP Synchronization: ESP.SYNCH / ESP.ADDITIONS
  - The ESP Synchronization ratio calculates the ratio of ESP explicit use (for example by load or store instruction) and implicit uses (for example by PUSH or POP instruction). The expected ratio value is 0.2 or lower. If the ratio is higher, consider rearranging your code to avoid ESP synchronization events.

22.8.1.4 Macro-fusion

- Macro-Fusion: UOPS_RETIRED.MACRO_FUSION / INST_RETIRED.ANY
  - The Macro-Fusion ratio calculates how many of the retired instructions were fused to a single micro-op. You may find this ratio is high for a 32-bit binary executable but significantly lower for the equivalent 64-bit binary, and the 64-bit binary performs slower than the 32-bit binary. A possible reason is the 32-bit binary benefited from macro-fusion significantly.

22.8.1.5 Length Changing Prefix (LCP) Stalls

- LCP Delays Detected: ILD_STALL / CPU_CLK_UNHALTED.CORE
  - A high value of the LCP Delays Detected ratio indicates that many Length Changing Prefix (LCP) delays occur in the measured code.

22.8.1.6 Self Modifying Code Detection

5. Self Modifying Code Clear Performance Impact: MACHINE_NUKES.SMC * 150 / CPU_CLK_UNHALTED.CORE * 100

A program that writes into code sections and shortly afterwards executes the generated code may incur severe penalties. Self Modifying Code Performance Impact estimates the percentage of cycles that the program spends on self-modifying code penalties.

22.8.2 BRANCH PREDICTION RATIOS

Appendix 22.8.1.2 discusses branching that impacts the front end performance. This section describes event ratios that are commonly used to characterize branch mispredictions.

22.8.2.1 Branch Mispredictions

- Branch Misprediction Performance Impact: RESOURCE_STALLS.BR_MISS_CLEAR / CPU_CLK_UNHALTED.CORE * 100
  - With the Branch Misprediction Performance Impact, you can tell the percentage of cycles that the processor spends in recovering from branch mispredictions.
- Branch Misprediction per Micro-Op Retired: BR_INST_RETIRED.MISPRED/UOPS_RETIRED.ANY
  - The ratio Branch Misprediction per Micro-Op Retired indicates if the code suffers from many branch mispredictions. In this case, improving the predictability of branches can have a noticeable impact on the performance of your code.
  - In addition, the performance impact of each branch misprediction might be high. This happens if the code prior to the mispredicted branch has high CPI, such as cache misses, which cannot be parallelized with following code due to the branch misprediction. Reducing the CPI of this code will reduce the misprediction performance impact. See other ratios to identify these cases.

You can use the precise event BR_INST_RETIRED.MISPRED to detect the actual targets of the mispredicted branches. This may help you to identify the mispredicted branch.

22.8.2.2 Virtual Tables and Indirect Calls

- Virtual Table Usage: BR_IND_CALL_EXEC / INST_RETIRED.ANY
  - A high value for the ratio Virtual Table Usage indicates that the code includes many indirect calls. The destination address of an indirect call is hard to predict.
- Virtual Table Misuse: BR_CALL_MISSP_EXEC / BR_INST_RETIRED.MISPRED
  - A high value of Branch Misprediction Performance Impact ratio (Ratio 15) together with high Virtual Table Misuse ratio indicate that significant time is spent due to mispredicted indirect function calls.
  - In addition to explicit use of function pointers in C code, indirect calls are used for implementing inheritance, abstract classes, and virtual methods in C++.

22.8.2.3 Mispredicted Returns

- Mispredicted Return Instruction Rate: BR_RET_MISSP_EXEC/BR_RET_EXEC
  - The processor has a special mechanism that tracks CALL-RETURN pairs. The processor assumes that every CALL instruction has a matching RETURN instruction. If a RETURN instruction restores a return address, which is not the one stored during the matching CALL, the code incurs a misprediction penalty.

22.8.3 EXECUTION RATIOS

This section covers event ratios that can provide insights to the interactions of micro-ops with RS, ROB, execution units, and so forth.

22.8.3.1 Resource Stalls

A high value for the RS Full Ratio (Ratio 2) indicates that the Reservation Station (RS) often gets full with micro-ops due to long dependency chains. The micro-ops that get into the RS cannot execute because they wait for their operands to be computed by previous micro-ops, or they wait for a free execution unit to be executed. This prevents exploiting the parallelism provided by the multiple execution units.

A high value for the ROB Full Ratio (Ratio 3) indicates that the reorder buffer (ROB) often gets full with micro-ops. This usually implies on long latency operations, such as L2 cache demand misses.

22.8.3.2 ROB Read Port Stalls

- ROB Read Port Stall Rate: RAT_STALLS.ROB_READ_PORT / CPU_CLK_UNHALTED.CORE
  - The ratio ROB Read Port Stall Rate identifies ROB read port stalls. However it should be used only if the number of resource stalls, as indicated by Resource Stall Ratio, is low.

22.8.3.3 Partial Register Stalls

- Partial Register Stalls Ratio: RAT_STALLS.PARTIAL_CYCLES / CPU_CLK_UNHALTED.CORE*100
  - Frequent accesses to registers that cause partial stalls increase access latency and decrease performance. Partial Register Stalls Ratio is the percentage of cycles when partial stalls occur.

22.8.3.4 Partial Flag Stalls

- Partial Flag Stalls Ratio:RAT_STALLS.FLAGS / CPU_CLK_UNHALTED.CORE
  - Partial flag stalls have high penalty and they can be easily avoided. However, in some cases, Partial Flag Stalls Ratio might be high although there are no real flag stalls. There are a few instructions that partially modify the RFLAGS register and may cause partial flag stalls. The most popular are the shift instructions (SAR, SAL, SHR, and SHL) and the INC and DEC instructions.

22.8.3.5 Bypass Between Execution Domains

- Delayed Bypass to FP Operation Rate: DELAYED_BYPASS.FP / CPU_CLK_UNHALTED.CORE
- Delayed Bypass to SIMD Operation Rate: DELAYED_BYPASS.SIMD / CPU_CLK_UNHALTED.CORE
- Delayed Bypass to Load Operation Rate: DELAYED_BYPASS.LOAD / CPU_CLK_UNHALTED.CORE
- Domain bypass adds one cycle to instruction latency. To identify frequent domain bypasses in the code you can use the above ratios.

22.8.3.6 Floating-Point Performance Ratios

- Floating-Point Instructions Ratio: X87_OPS_RETIRED.ANY / INST_RETIRED.ANY * 100

Significant floating-point activity indicates that specialized optimizations for floating-point algorithms may be applicable.

- FP Assist Performance Impact: FP_ASSIST * 80 / CPU_CLK_UNHALTED.CORE * 100

  - Floating-Point assist is activated for non-regular FP values like denormals and NANs. FP assist is extremely slow compared to regular FP execution. Different assists incur different penalties. FP Assist Performance Impact estimates the overall impact.
- Divider Busy: IDLE_DURING_DIV / CPU_CLK_UNHALTED.CORE * 100
  - A high value for the Divider Busy ratio indicates that the divider is busy and no other execution unit or load operation is in progress for many cycles. Using this ratio ignores L1 data cache misses and L2 cache misses that can be executed in parallel and hide the divider penalty.
- Floating-Point Control Word Stall Ratio: RESOURCE_STALLS.FPCW / CPU_CLK_UNHALTED.CORE * 100
  - Frequent modifications to the Floating-Point Control Word (FPCW) might significantly decrease performance. The main reason for changing FPCW is for changing rounding mode when doing FP to integer conversions.

22.8.4 MEMORY SUB-SYSTEM - ACCESS CONFLICTS RATIOS

A high value for Load or Store Buffer Full Ratio (Ratio 4) indicates that the load buffer or store buffer are frequently full, hence new micro-ops cannot enter the execution pipeline. This can reduce execution parallelism and decrease performance.

- Load Rate: L1D_CACHE_LD.MESI / CPU_CLK_UNHALTED.CORE

One memory read operation can be served by a core each cycle. A high “Load Rate” indicates that execution may be bound by memory read operations.

6. Store Order Block: STORE_BLOCK.ORDER / CPU_CLK_UNHALTED.CORE * 100

Store Order Block ratio is the percentage of cycles that store operations, which miss the L2 cache, block committing data of later stores to the memory sub-system. This behavior can further cause the store buffer to fill up (see Ratio 4).

22.8.4.1 Loads Blocked by the L1 Data Cache

7. Loads Blocked by L1 Data Cache Rate: LOAD_BLOCK.L1D/CPU_CLK_UNHALTED.CORE

A high value for “Loads Blocked by L1 Data Cache Rate” indicates that load operations are blocked by the L1 data cache due to lack of resources, usually happening as a result of many simultaneous L1 data cache misses.

22.8.4.2 4K Aliasing and Store Forwarding Block Detection

8. Loads Blocked by Overlapping Store Rate: LOAD_BLOCK.OVERLAP_STORE/CPU_CLK_UNHALTED.CORE

4K aliasing and store forwarding block are two different scenarios in which loads are blocked by preceding stores due to different reasons. Both scenarios are detected by the same event: LOAD_BLOCK.OVERLAP_STORE. A high value for “Loads Blocked by Overlapping Store Rate” indicates that either 4K aliasing or store forwarding block may affect performance.

22.8.4.3 Load Block by Preceding Stores

9. Loads Blocked by Unknown Store Address Rate: LOAD_BLOCK.STA / CPU_CLK_UNHALTED.CORE

A high value for “Loads Blocked by Unknown Store Address Rate” indicates that loads are frequently blocked by preceding stores with unknown address and implies performance penalty.

10. Loads Blocked by Unknown Store Data Rate: LOAD_BLOCK.STD / CPU_CLK_UNHALTED.CORE

A high value for “Loads Blocked by Unknown Store Data Rate” indicates that loads are frequently blocked by preceding stores with unknown data and implies performance penalty.

22.8.4.4 Memory Disambiguation

The memory disambiguation feature of Intel Core microarchitecture uses a predictor to allow loads to execute speculatively in the presence of older unknown stores. This eliminates most of the non-required load blocks by stores with an unknown address. The LOAD_BLOCK.STA and MEMORY_DISAMBIGUATION.RESET events can be used to measure the effectiveness of the feature.

22.8.4.5 Load Operation Address Translation

11. L0 DTLB Miss due to Loads - Performance Impact: DTLB_MISSES.L0_MISS_LD * 2 / CPU_CLK_UNHALTED.CORE

High number of DTLB0 misses indicates that the data set that the workload uses spans a number of pages that is bigger than the DTLB0. The high number of misses is expected to impact workload performance only if the CPI (Ratio 1) is low - around 0.8. Otherwise, it is likely that the DTLB0 miss cycles are hidden by other latencies.

22.8.5 MEMORY SUB-SYSTEM - CACHE MISSES RATIOS

22.8.5.1 Locating Cache Misses in the Code

Intel Core microarchitecture provides you with precise events for retired load instructions that miss the L1 data cache or the L2 cache. As precise events they provide the instruction pointer of the instruction following the one that caused the event. Therefore the instruction that comes immediately prior to the pointed instruction is the one that causes the cache miss. These events are most helpful to quickly identify on which loads to focus to fix a performance problem. The events are:

- MEM_LOAD_RETIRE.L1D_MISS
- MEM_LOAD_RETIRE.L1D_LINE_MISS
- MEM_LOAD_RETIRE.L2_MISS
- MEM_LOAD_RETIRE.L2_LINE_MISS

22.8.5.2 L1 Data Cache Misses

12. L1 Data Cache Miss Rate: L1D_REPL / INST_RETIRED.ANY

A high value for L1 Data Cache Miss Rate indicates that the code misses the L1 data cache too often and pays the penalty of accessing the L2 cache. See also Loads Blocked by L1 Data Cache Rate (Ratio 32).

You can count separately cache misses due to loads, stores, and locked operations using the events L1D_CACHE_LD.I_STATE, L1D_CACHE_ST.I_STATE, and L1D_CACHE_LOCK.I_STATE, accordingly.

22.8.5.3 L2 Cache Misses

13. L2 Cache Miss Rate: L2_LINES_IN.SELF.ANY / INST_RETIRED.ANY

A high L2 Cache Miss Rate indicates that the running workload has a data set larger than the L2 cache. Some of the data might be evicted without being used. Unless all the required data is brought ahead of time by the hardware prefetcher or software prefetching instructions, bringing data from memory has a significant impact on the performance.

14. L2 Cache Demand Miss Rate: L2_LINES_IN.SELF.DEMAND / INST_RETIRED.ANY

A high value for L2 Cache Demand Miss Rate indicates that the hardware prefetchers are not exploited to bring the data this workload needs. Data is brought from memory when needed to be used and the workload bears memory latency for each such access.

22.8.6 MEMORY SUB-SYSTEM - PREFETCHING

22.8.6.1 L1 Data Prefetching

The event L1D_PREFETCH.REQUESTS is counted whenever the DCU attempts to prefetch cache lines from the L2 (or memory) to the DCU. If you expect the DCU prefetchers to work and to count this event, but instead you detect the event MEM_LOAD_RETIRE.L1D_MISS, it might be that the IP prefetcher suffers from load instruction address collision of several loads.

22.8.6.2 L2 Hardware Prefetching

With the event L2_LD.SELF.PREFETCH.MESI you can count the number of prefetch requests that were made to the L2 by the L2 hardware prefetchers. The actual number of cache lines prefetched to the L2 is counted by the event L2_LD.SELF.PREFETCH.I_STATE.

22.8.6.3 Software Prefetching

The events for software prefetching cover each level of prefetching separately.

15. Useful PrefetchT0 Ratio: SSE_PRE_MISS.L1 / SSE_PRE_EXEC.L1 * 100

16. Useful PrefetchT1 and PrefetchT2 Ratio: SSE_PRE_MISS.L2 / SSE_PRE_EXEC.L2 * 100

A low value for any of the prefetch usefulness ratios indicates that some of the SSE prefetch instructions prefetch data that is already in the caches.

17. Late PrefetchT0 Ratio: LOAD_HIT_PRE / SSE_PRE_EXEC.L1

18. Late PrefetchT1 and PrefetchT2 Ratio: LOAD_HIT_PRE / SSE_PRE_EXEC.L2

A high value for any of the late prefetch ratios indicates that software prefetch instructions are issued too late and the load operations that use the prefetched data are waiting for the cache line to arrive.

22.8.7 MEMORY SUB-SYSTEM - TLB MISS RATIOS

19. TLB miss penalty: PAGE_WALKS.CYCLES / CPU_CLK_UNHALTED.CORE * 100

A high value for the TLB miss penalty ratio indicates that many cycles are spent on TLB misses. Reducing the number of TLB misses may improve performance. This ratio does not include DTLB0 miss penalties (see Ratio 37).

The following ratios help to focus on the kind of memory accesses that cause TLB misses most frequently See “ITLB Miss Rate” (Ratio 6) for TLB misses due to instruction fetch.

20. DTLB Miss Rate: DTLB_MISSES.ANY / INST_RETIRED.ANY

A high value for DTLB Miss Rate indicates that the code accesses too many data pages within a short time, and causes many Data TLB misses.

21. DTLB Miss Rate due to Loads: DTLB_MISSES.MISS_LD / INST_RETIRED.ANY

A high value for DTLB Miss Rate due to Loads indicates that the code accesses loads data from too many pages within a short time, and causes many Data TLB misses. DTLB misses due to load operations may have a significant impact, since the DTLB miss increases the load operation latency. This ratio does not include DTLB0 miss penalties (see Ratio 37).

To precisely locate load instructions that caused DTLB misses you can use the precise event MEM_LOAD_RETIRE.DTLB_MISS.

22. DTLB Miss Rate due to Stores: DTLB_MISSES.MISS_ST / INST_RETIRED.ANY

A high value for DTLB Miss Rate due to Stores indicates that the code accesses too many data pages within a short time, and causes many Data TLB misses due to store operations. These misses can impact performance if they do not occur in parallel to other instructions. In addition, if there are many stores in a row, some of them missing the DTLB, it may cause stalls due to full store buffer.

22.8.8 MEMORY SUB-SYSTEM - CORE INTERACTION

22.8.8.1 Modified Data Sharing

23. Modified Data Sharing Ratio: EXT_SNOOP.ALL_AGENTS.HITM / INST_RETIRED.ANY

Frequent occurrences of modified data sharing may be due to two threads using and modifying data laid in one cache line. Modified data sharing causes L2 cache misses. When it happens unintentionally (aka false sharing) it usually causes demand misses that have high penalty. When false sharing is removed code performance can dramatically improve.

24. Local Modified Data Sharing Ratio: EXT_SNOOP.THIS_AGENT.HITM / INST_RETIRED.ANY

Modified Data Sharing Ratio indicates the amount of total modified data sharing observed in the system. For systems with several processors you can use Local Modified Data Sharing Ratio to indicates the amount of modified data sharing between two cores in the same processor. (In systems with one processor the two ratios are similar).

22.8.8.2 Fast Synchronization Penalty

25. Locked Operations Impact: (L1D_CACHE_LOCK_DURATION + 20 * L1D_CACHE_LOCK.MESI) / CPU_CLK_UNHALTED.CORE * 100

Fast synchronization is frequently implemented using locked memory accesses. A high value for Locked Operations Impact indicates that locked operations used in the workload have high penalty. The latency of a locked operation depends on the location of the data: L1 data cache, L2 cache, other core cache or memory.

22.8.8.3 Simultaneous Extensive Stores and Load Misses

26. Store Block by Snoop Ratio: (STORE_BLOCK.SNOOP / CPU_CLK_UNHALTED.CORE) * 100

A high value for “Store Block by Snoop Ratio” indicates that store operations are frequently blocked and performance is reduced. This happens when one core executes a dense stream of stores while the other core in the processor frequently snoops it for cache lines missing in its L1 data cache.

22.8.9 MEMORY SUB-SYSTEM - BUS CHARACTERIZATION

22.8.9.1 Bus Utilization

27. Bus Utilization: BUS_TRANS_ANY.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Utilization is the percentage of bus cycles used for transferring bus transactions of any type. In single processor systems most of the bus transactions carry data. In multiprocessor systems some of the bus transactions are used to coordinate cache states to keep data coherency.

28. Data Bus Utilization: BUS_DRDY_CLOCKS.ALL_AGENTS / CPU_CLK_UNHALTED.BUS * 100

Data Bus Utilization is the percentage of bus cycles used for transferring data among all bus agents in the system, including processors and memory. High bus utilization indicates heavy traffic between the processor(s) and memory. Memory sub-system latency can impact the performance of the program. For compute-intensive applications with high bus utilization, look for opportunities to improve data and code locality. For other types of applications (for example, copying large amounts of data from one memory area to another), try to maximize bus utilization.

29. Bus Not Ready Ratio: BUS_BNR_DRV.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS * 100

Bus Not Ready Ratio estimates the percentage of bus cycles during which new bus transactions cannot start. A high value for Bus Not Ready Ratio indicates that the bus is highly loaded. As a result of the Bus Not Ready (BNR) signal, new bus transactions might defer and their latency will have higher impact on program performance.

30. Burst Read in Bus Utilization: BUS_TRANS_BRD.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

A high value for Burst Read in Bus Utilization indicates that bus and memory latency of burst read operations may impact the performance of the program.

31. RFO in Bus Utilization: BUS_TRANS_RFO.SELF * 2 / CPU_CLK_UNHALTED.BUS * 100

A high value for RFO in Bus Utilization indicates that latency of Read For Ownership (RFO) transactions may impact the performance of the program. RFO transactions may have a higher impact on the program performance compared to other burst read operations (for example, as a result of loads that missed the L2). See also Ratio 31.

22.8.9.2 Modified Cache Lines Eviction

32. L2 Modified Lines Eviction Rate: L2_M_LINES_OUT.SELF.ANY / INST_RETIRED.ANY

When a new cache line is brought from memory, an existing cache line, possibly modified, is evicted from the L2 cache to make space for the new line. Frequent evictions of modified lines from the L2 cache increase the latency of the L2 cache misses and consume bus bandwidth.

33. Explicit WB in Bus Utilization: BUS_TRANS_WB.SELF * 2 / CPU_CLK_UNHALTED.BUS*100

Explicit Write-back in Bus Utilization considers modified cache line evictions not only from the L2 cache but also from the L1 data cache. It represents the percentage of bus cycles used for explicit write-backs from the processor to memory.

APPENDIX A
APPLICATION PERFORMANCE TOOLS

Intel offers an array of application performance tools that are optimized to take advantage of the Intel architecture (IA)-based processors. This appendix introduces these tools and explains their capabilities for developing the most efficient programs without having to write assembly code.

The following performance tools are available.

- **Compilers**
  - Intel® C++ Compiler: a high-performance, optimized C and C++ cross compiler with the capability of offloading compute-intensive code to Intel® Many Integrated Core Architecture (Intel® MIC Architecture) as well as Intel® HD Graphics, and executing on multiple execution units by using Intel® Cilk™ parallel extensions.
  - Intel® Fortran Compiler: a high-performance, optimized Fortran compiler.
- **Performance Libraries** — a set of software libraries optimized for Intel architecture processors.
  - Intel® Integrated Performance Primitives (Intel® IPP): performance building blocks to boost embedded system performance.
  - Intel® Math Kernel Library (Intel® MKL): a set of highly optimized linear algebra, Fast Fourier Transform (FFT), vector math, and statistics functions.
  - Intel® Threading Building Blocks (Intel® TBB): a C and C++ template library for creating high performance, scalable parallel applications.
  - Intel® Data Analytics Acceleration Library (Intel® DAAL): C++ and Java API library of optimized analytics building blocks for all data analysis stages, from data acquisition to data mining and machine learning. Essential for engineering high performance Big Data applications.
- **Performance Profilers** — performance profilers collect, analyze, and display software performance data for tuning CPU, GPU, threading, vectorization and MPI parallelism from the system-wide view down to a specific line of code.
  - Intel® VTune™ Amplifier XE: performance profiler.
  - Intel® Graphics Performance Analyzers (Intel® GPA) - a set of performance analyzers for graphics applications.
  - Intel® Advisor: vectorization optimization and thread prototyping.
  - Intel® Trace Analyzer and Collector: MPI communications performance profiler and correctness checker.
- **Debuggers**
  - Intel® Inspector: memory and thread debugger.
  - Intel® Application Debugger.
  - Intel® JTAG Debugger.
  - Intel® System Debugger.
- **Cluster Tools**
  - Intel® MPI Library: high-performance MPI library.
  - Intel® MPI Benchmarks: a set of MPI kernel tests to verify the performance of your cluster or MPI implementation.

The performance tools listed above can be found in the following product suites.

- **Intel® Parallel Studio XE**[1]
  - Intel® Media Server Studio.
  - Intel® Systems Studio.

A.1 COMPILERS

Intel compilers support several general optimization settings, including /O1, /O2, /O3, and /fast. Each of them enables a number of specific optimization options. In most cases, /O2 is recommended over /O1 because the /O2 option enables function expansion, which helps programs that have many calls to small functions. The /O1 may sometimes be preferred when code size is a concern. The /O2 option is on by default.

The /Od (-O0 on Linux) option disables all optimizations. The /O3 option enables more aggressive optimizations, most of which are effective only in conjunction with processor-specific optimizations described below.

The /fast option maximizes speed across the entire program. For most Intel 64 and IA-32 processors, the "/fast" option is equivalent to "/O3 /Qipo /Qprec-div- /fp:fast=2 /QxHost" on Windows*, "-ipo -O3 -no-prec-div -static -fp-model fast=2 -xHost" on Linux, and "-ipo -mdynamic-no-pic -O3 -no-prec-div -fp-model fast=2 -xHost" on OS X*.

All the command-line options are described in *Intel® C++ Compiler documentation*.

A.1.1 RECOMMENDED OPTIMIZATION SETTINGS FOR INTEL® 64 AND IA-32 PROCESSORS

Table A-1 lists some examples of recommended compiler options for generating code for Intel processors. Table A-1 also applies to code targeted to run in compatibility mode on an Intel 64 processor, but does not apply to running in 64-bit mode.

Table A-1. Recommended Processor Optimization Options

Need	Recommendation	Comments
Best performance on Intel processors utilizing Intel® AVX2 instructions.	• /QxCORE-AVX2 (-x CORE-AVX2 on Linux and Mac OS)	• Single code path.
Best performance on Intel processors utilizing Intel® AVX2 instructions.	• /QaxCORE-AVX2 (-ax CORE-AVX2 on Linux and Mac OS)	• Multiple code paths are generated. • Be sure to validate your application on all systems where it may be deployed.
Best performance on Intel processors utilizing Intel SSE4.2 instructions.	• /QxSSE4.2 (-xSSE4.2 on Linux and Mac OS)	• Single code path.
Best performance on Intel processors utilizing Intel SSE4.2 instructions.	• /QaxSSE4.2 (-axSSE4.2 on Linux and Mac OS)	• Multiple code paths are generated. • Be sure to validate your application on all systems where it may be deployed.

1. Details on versions and tools included can be found here: https://software.intel.com/en-us/intel-parallel-studio-xe.

A.1.2 VECTORIZATION AND LOOP OPTIMIZATION

The Intel C++ and Fortran Compiler's vectorization feature can detect sequential data access by the same instruction and transforms the code to use Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3 and Intel SSE4, depending on the target processor platform. The vectorizer supports the following features:

- Multiple data types: Float/double, char/short/int/long (both signed and unsigned), _Complex float/double are supported.
- Step by step diagnostics: Through the /Qopt-report /Qopt-report-phase (-qopt-report -qopt-report-phase on Linux and Mac OS) switch, the vectorizer can identify, line-by-line and variable-by-variable, what code was vectorized, what code was not vectorized, and more importantly, why it was not vectorized. This feedback gives the developer the information necessary to slightly adjust or restructure code, with dependency directives and restrict keywords, to allow vectorization to occur.
- Advanced dynamic data-alignment strategies: Alignment strategies include loop peeling and loop unrolling. Loop peeling can generate aligned loads, enabling faster application performance. Loop unrolling matches the prefetch of a full cache line and allows better scheduling.
- Portable code: By using appropriate Intel compiler switches to take advantage new processor features, developers can avoid the need to rewrite source code.

The processor-specific vectorizer switch options are: /Qx<CODE> and /Qax<CODE> (-x<CODE> and -xa<CODE> on Linux and Mac OS). The compiler provides a number of other vectorizer switch options that allow you to control vectorization. The latter switches require one of these switches to be on. The default is off.

A.1.2.1 Multithreading with OpenMP*

Both the Intel C++ and Fortran Compilers support shared memory parallelism using OpenMP compiler directives, library functions and environment variables. OpenMP directives are activated by the compiler switch /Qopenmp (-openmp on Linux and Mac OS). The available directives are described in the Compiler User's Guides available with the Intel C++ and Fortran Compilers. For information about the OpenMP standard, see http://www.openmp.org.

A.1.2.2 Automatic Multithreading

Both the Intel C++ and Fortran Compilers can generate multithreaded code automatically for simple loops with no dependencies. This is activated by the compiler switch /Qparallel (-parallel in Linux and Mac OS).

A.1.3 INLINE EXPANSION OF LIBRARY FUNCTIONS (/OI, /OI-)

The compiler inlines a number of standard C, C++, and math library functions by default. This usually results in faster execution. Sometimes, however, inline expansion of library functions can cause unexpected results. For explanation, see the Intel C++ Compiler documentation.

A.1.4 INTERPROCEDURAL AND PROFILE-GUIDED OPTIMIZATIONS

The following are two methods to improve the performance of your code based on its unique profile and procedural dependencies.

A.1.4.1 Interprocedural Optimization (IPO)

You can use the **/Qip** (-ip in Linux and Mac OS) option to analyze your code and apply optimizations between procedures within each source file. Use multifile IPO with /Qipo (-ipo in Linux and Mac OS) to enable the optimizations between procedures in separate source files.

A.1.4.2 Profile-Guided Optimization (PGO)

Creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the compiler generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Profile-guided optimization is particularly beneficial for the Pentium 4 and Intel Xeon processor family. It greatly enhances the optimization decisions the compiler makes regarding instruction cache utilization and memory paging. Also, because PGO uses execution-time information to guide the optimizations, branch-prediction can be significantly enhanced by reordering branches and basic blocks to keep the most commonly used paths in the microarchitecture pipeline, as well as generating the appropriate branch-hints for the processor.

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

NOTE

The compiler issues a warning that the dynamic information corresponds to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

For more on code optimization options, see the Intel C++ Compiler documentation.

A.1.5 INTEL® CILK™ PLUS

Intel Cilk Plus is an Intel C/C++ compiler extension with only 3 keywords that simplifies implementing simple loop and task parallel applications. It offers superior functionality by combining vectorization features with high-level loop-type data parallelism and tasking.

A.2 PERFORMANCE LIBRARIES

The Intel Performance Libraries implement a number of optimizations that are discussed throughout this manual. Examples include architecture-specific tuning such as loop unrolling, instruction pairing and scheduling; and memory management with explicit and implicit data prefetching and cache tuning.

The Libraries take advantage of the parallelism in the SIMD instructions using MMX technology, Intel Streaming SIMD Extensions (Intel SSE), Intel Streaming SIMD Extensions 2 (Intel SSE2), and Intel Streaming SIMD Extensions 3 (Intel SSE3). These techniques improve the performance of computationally intensive algorithms and deliver hand coded performance in a high level language development environment.

For performance sensitive applications, the Intel Performance Libraries free the application developer from the time consuming task of assembly-level programming for a multitude of frequently used functions. The time required for prototyping and implementing new application features is substantially reduced and most important, the time to market is substantially improved. Finally, applications developed with the Intel Performance Libraries benefit from

new architectural features of future generations of Intel processors simply by relinking the application with upgraded versions of the libraries.

The library set includes the Intel Integrated Performance Primitives (Intel IPP), Intel Math Kernel Library (Intel MKL) and Intel Threading Building Blocks (Intel TBB).

A.2.1 INTEL® INTEGRATED PERFORMANCE PRIMITIVES (INTEL® IPP)

Intel Integrated Performance Primitives for Linux and Windows: IPP is a cross-platform software library which provides a range of library functions for video decode/encode, audio decode/encode, image color conversion, computer vision, data compression, string processing, signal processing, image processing, JPEG decode/encode, speech recognition, speech decode/encode, cryptography plus math support routines for such processing capabilities.

These ready-to-use functions are highly optimized using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instruction sets. With a single API across the range of platforms, the users can have platform compatibility and reduced cost of development.

A.2.2 INTEL® MATH KERNEL LIBRARY (INTEL® MKL)

The Intel Math Kernel Library for Linux, Windows and OS X: MKL is composed of highly optimized mathematical functions for engineering, scientific and financial applications requiring high performance on Intel platforms. The functional areas of the library include linear algebra consisting of LAPACK and BLAS, Discrete Fourier Transforms (DFT), vector transcendental functions (vector math library/VML) and vector statistical functions (VSL). Intel MKL is optimized for the latest features and capabilities of the Intel® Itanium® , Intel® Xeon®, Intel® Pentium® 4, and Intel® Core2 Duo processor-based systems. Special attention has been paid to optimizing multi-threaded performance for the new Quad-Core Intel® Xeon® processor 5300 series.

A.2.3 INTEL® THREADING BUILDING BLOCKS (INTEL® TBB)

Intel TBB is a C++ template library for creating reliable, portable, and scalable parallel applications. Use Intel TBB for a simple and rapid way of developing robust task-based parallel applications that scale to available processor cores, are compatible with multiple environments, and are easier to maintain.

Intel TBB is validated and commercially supported on Windows, Linux and OS X* platforms. It is also available on FreeBSD*, IA Solaris*, XBox* 360, and PowerPC-based systems via the open source community.

A.2.4 BENEFITS SUMMARY

The overall benefits the libraries provide to the application developers are as follows:

- **Time-to-Market**: Low-level building block functions that support rapid application development, improving time to market.
- **Performance:** Highly-optimized routines with a C interface that give Assembly-level performance in a C/C++ development environment (Intel MKL also supports a Fortran interface).
- **Platform tuned:** Processor-specific optimizations that yield the best performance for each Intel processor.
- **Compatibility:** Processor-specific optimizations with a single application programming interface (API) to reduce development costs while providing optimum performance.
- **Threaded application support:** Applications can be threaded with the assurance that the Intel MKL and Intel IPP functions are safe for use in a threaded environment.

A.3 PERFORMANCE PROFILERS

Intel® serial and parallel processing profiling tools locate performance bottlenecks without recompilation and with very low overhead, and provide quick access to scaling information for faster and improved decision making. The profiling tools enable evaluation of all sizes of Intel® processor based systems, from embedded systems through supercomputers, to help you improve application performance.

A.3.1 INTEL® VTUNE™ AMPLIFIER XE

Intel® VTune™ Amplifier XE[1] is a powerful threading and performance optimization tool for Windows and Linux. Use the VTune Amplifier to fine-tune for optimal performance, ensuring cores are fully exploited and new processor capabilities are supported to the fullest.

The sections that follow briefly describe the major features of the VTune Amplifier. For more details on these features, run the VTune Amplifier and see the online documentation.

A.3.1.1 Hardware Event-Based Sampling Analysis

VTune Amplifier introduces a set of microarchitecture analysis types based on the event-based sampling data collection and targeted for the Intel® Core™ 2 processor family, processors based on the Intel processors. Depending on the analysis type, the VTune Amplifier monitors a set of hardware events and displays collected data as raw event count (for example, cache misses, clock ticks, and instructions retired) and as performance metrics. Each metric is an event ratio with its own threshold values. As soon as the performance of a program unit per metric exceeds the threshold, the VTune Amplifier marks this value as a performance issue (in pink) and provides recommendations how to fix it.[2]

A.3.1.2 Algorithm Analysis

VTune Amplifier introduces a set of algorithm analysis types based on the user-mode sampling and tracing collection:

- **Basic Hotspots** analysis that helps understand the application execution flow and identify sections of code that took a long time to execute (hotspots). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hotspots can be removed, while other hotspots are fundamental to the application functionality and cannot be removed. VTune Amplifier creates a list of functions in your application ordered by the amount of time spent in a function. It also detects the call stacks for each of these functions so you can see how the hot functions are called.
- **Locks and Waits** analysis that helps identify the cause of the ineffective processor utilization. One of the most common problems is threads waiting too long on synchronization objects (locks). Performance suffers when waits occur while cores are under-utilized. During the Locks and Waits analysis you can estimate the impact each synchronization object introduces to the application and understand how long the application was required to wait on each synchronization object, or in blocking APIs, such as sleep and blocking I/O.
- **Concurrency** analysis that helps identify hotspot functions where processor utilization is poor. When cores are idle at a hotspot, you have an opportunity to improve performance by getting those cores working for you.

---

1. For additional information, see: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/?wapkw=vtune
2. *Lists of available performance-monitoring events can be found at: https://perfmon-events.intel.com/.*

A.3.1.3 Platform Analysis

You may enable the VTune Amplifier to collect platform-wide metrics for applications that use a Graphics Processing Unit (GPU) for rendering, video processing, and computations. Use the CPU/GPU Concurrency analysis as a starting point to understand the code execution on the various CPU and GPU cores in your system and identify whether your target application is GPU or CPU bound.

A.4 THREAD AND MEMORY CHECKERS

Intel® tools combine threading and memory error checking into one powerful error checking tool to help increase the reliability, security, and accuracy of your applications.

A.4.1 INTEL® INSPECTOR

Intel® Inspector provides thread debugging analysis for higher performing parallel applications (find: data races, deadlocks, thread and sync APIs used and memory accesses between threads) and memory checking analysis for serial and parallel applications (find: memory leaks and memory corruption, memory allocation and deallocation API mismatches, and inconsistent memory API usage).

Intel® Inspector enhances developer productivity and facilitates application reliability by effectively finding crucial memory and threading defects early in the development cycle. It gives detailed insights into application memory and threading behavior to improve application reliability. The powerful thread checker and debugger makes it easier to find latent errors on the executed code path. It also finds intermittent and non-deterministic errors, even if the error-causing timing scenario does not happen. In addition, developers can test their code more often, without the need to use special test builds or compilers.

A.5 VECTORIZATION ASSISTANT

A.5.1 INTEL® ADVISOR

The Intel Advisor is the vectorization assistant and threading prototyping tool that simplifies threading, parallelizing, and vectorizing your source code by identifying those areas in your applications where vectorization and/or threading parallelism would have the greatest impact.

A.6 CLUSTER TOOLS

The Intel Parallel Studio XE, Cluster Edition helps you develop, analyze and optimize performance of parallel applications for clusters using IA-32, IA-64, and Intel® 64 architectures. The Cluster Edition includes the following tools for developing code for clusters: Intel® Trace Analyzer and Collector, Intel MPI Library, and Intel MPI Benchmarks.

A.6.1 INTEL® TRACE ANALYZER AND COLLECTOR

The Intel® Trace Analyzer and Collector[1] helps to provide information critical to understanding and optimizing application performance on clusters by quickly finding performance bottlenecks in MPI communication. It supports

1. Intel® Trace Analyzer and Collector is only available as part of Intel® Cluster Studio or Intel® Cluster Studio XE.

Intel® architecture-based cluster systems, features a high degree of compatibility with current standards, and includes trace file idealization and comparison, counter data displays, performance assistant and an MPI correctness checking library. Analyze MPI performance, speed up parallel application runs, locate hotspots and bottlenecks, and compare trace files with graphics providing extensively detailed analysis and aligned timelines.

A.6.1.1 MPI Performance Snapshot

The MPI Performance Snapshot (MPS) is a scalable lightweight performance tool for MPI applications. It collects a variety of MPI application statistics (such as communication, activity, and load balance) and presents it in an easy-to-read format. MPS combines lightweight statistics from the Intel® MPI Library with OS and hardware-level counters to provide you with high-level overview of your application. The tool is provided as part of the Intel® Trace Analyzer and Collector installation.

A.6.2 INTEL® MPI LIBRARY

The Intel MPI Library is a multi-fabric message passing library that implements the Message Passing Interface, v2 (MPI-2) specification. It provides a standard library across Intel® platforms. The Intel MPI Library supports multiple hardware fabrics including InfiniBand, Myrinet*, and Intel® True Scale Fabric. Intel® MPI Library covers all your configurations by providing an accelerated universal, multi-fabric layer for fast interconnects via the Direct Access Programming Library (DAPL) methodology. Develop MPI code independent of the fabric, knowing it will run efficiently on whatever fabric is chosen by the user at runtime.

Intel MPI Library dynamically establishes the connection, but only when needed, which reduces the memory footprint. It also automatically chooses the fastest transport available. The fallback to sockets at job startup avoids the chance of execution failure even if the interconnect selection fails. This is especially helpful for batch computing. Any products developed with Intel MPI Library are assured run time compatibility since your users can download Intel's free runtime environment kit. Application performance can also be increased via the large message bandwidth advantage from the optional use of DAPL inside a multi-core or SMP node.

A.6.3 INTEL® MPI BENCHMARKS

The Intel MPI Benchmarks will help enable an easy performance comparison of MPI functions and patterns, the benchmark features improvements in usability, application performance, and interoperability.

A.7 INTEL® COMMUNITIES

You can find information on classroom training offered by Intel on the Intel Community Page. Find general information for developers on Intel's Developer Zone.

APPENDIX B
RUNTIME PERFORMANCE OPTIMIZATION BLUEPRINT: INTEL® ARCHITECTURE OPTIMIZATION WITH LARGE CODE PAGES

This appendix provides a runtime optimization blueprint illustrating how the performance of runtimes can be improved by using large code pages.

B.1 OVERVIEW

Modern microprocessors support multiple page sizes for program code. Intel platforms have supported 4 KB and 2 MB pages for instructions as far back as 2011 in the Intel® Xeon® E5 processor (based on Ivy Bridge microarchitecture). Nevertheless, most programs use only one page size, which is the default of 4 KB. On Linux*, all applications are loaded into 4 KB memory pages by default. When examining performance bottlenecks for workloads on language runtimes, high stalls due to ITLB misses are found. This is largely due to the runtimes using only 4 KB pages for instructions.

B.1.1 TLBS AND STALLS

Intel processors have a Translation Lookaside Buffer (TLB), which stores the most recently used page-directory and page-table entries. TLBs speed up memory accesses when paging is enabled, by reducing the number of memory accesses that are required to read the page tables stored in system memory.

The TLBs are divided into the following groups:

- Instruction TLBs for 4KB pages.
- Data TLBs for 4KB pages.
- Instruction TLBs for large pages (2MB, 4MB pages).
- Data TLBs for large pages (2MB, 4MB, or 1GB pages).

On the Intel® Xeon® Platinum 8180 processor (based on Skylake Server microarchitecture), each processor TLB consists of dedicated L1 TLB for instruction cache (ITLB). Additionally, there is a unified L2 Second Level TLB (STLB) which is shared across both data and instructions, as shown below.

TLBs:

- ITLB
  - 4 KB page translations:
    - **128 entries; 8-way set associative.**
    - **Dynamic partitioning.**
  - 2 MB / 4 MB page translations:
    - **8 entries per thread; fully associative.**
    - **Duplicated for each thread.**
- STLB
  - 4 KB + 2 MB page translations:
    - **1536 entries; 12-way set associative, fixed partition.**

When the processor does not find an entry in the ITLB, it has to do a page walk and populate the entry. A miss in the L1 (first level) ITLBs results in a very small penalty that can usually be hidden by the Out of Order (OOO) execution. A miss in the STLB results in the page walker being invoked; this penalty can be noticeable in the execution. During this process, the processor is stalled. The following table lists the TLB sizes across different Intel product generations.

Table B-1. Core TLB Structure Size and Organization Across Multiple Intel Product Generations

TLB	Sandy Bridge/ Ivy Bridge Microarchitecture	Haswell / Broadwell Microarchitecture	Skylake / Cascade Lake Microarchitecture
L1 Instruction TLB	4K – 128, 4-way 2M/4M – 8/thread	4K – 128, 4 way 2M/4M – 8/thread	4K – 128, 8 way 2M/4M – 8/thread
L1 Data TLB	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way	4K – 64, 4-way 2M/4M – 32 – 4-way 1G: 4, 4-way
L2 (Unified) STLB	4K – 512, 4-way	4K+2M shared: Haswell: 1024, 8-way Broadwell: 1536, 6-way 1G: 16, 4-way	4K+2M shared: 1536, 12-way 1G: 16, 4-way

From Table B-1 we can see that 2M page entries are shared in the L2 Unified TLB from Haswell microarchitecture onwards.

B.1.2 LARGE PAGES

Both Windows* and Linux allow server applications to establish large-page memory regions. Using large 2MB pages, 20MB of memory can be mapped with just 10 pages; whereas with 4KB pages, 5120 pages are required. This means fewer TLB entries are needed, in turn reducing the number of TLB misses. Large pages can be used for code or for data, or both. Large pages for data are good to try if your workload has large heap. The blueprint described here focuses on using large pages for code.

B.2 DIAGNOSING THE PROBLEM

B.2.1 ITLB MISSES

Intel has defined a Top-down Micro-architecture Analysis Method (TMAM) (see Chapter 22, "Using Performance Monitoring Events"), which proposes a hierarchical execution cycles breakdown based on a set of new performance events. TMAM examines every instruction issue slot independently, and is therefore able to provide an accurate slot-level breakdown.

One of the components of the front-end latency is the ITLB miss stall. This metric represents the fraction of cycles the processor was stalled due to instruction TLB misses. On the Intel® Xeon® Scalable family of processors (based on Skylake Server microarchitecture), ITLB miss stall can be computed through two PMU counters, ICACHE_64B.IFTAG_STALL and CPU_CLK_UNHALTED.THREAD, using Equation 1.

Equation 1: Calculation of the ITLB Stall Metric:

$$ITLB_Miss_{stall} = 100 * (\frac{ICACHE_64B.IFTAG_STALL}{CPU_CLK_UNHALTED.THREAD})$$

Let us look at a concrete example. The Ghost.js workload has an ITLB_miss stall % of 10.6 when run in cluster mode across the whole system. A sampling of these two counters along with Equation 1 enables us to determine the % of ITLB_miss stall. A 10.6% stall due to ITLB misses is significant for this workload.

Table B-2. Calculating ITLB Miss Stall for Ghost.js

CPU_CLK_UNHALTED.THREAD	69838983
ICACHE_64B.IFTAG_STALL	7412534
ITLB Miss Stall %	10.6

Measuring ITLB miss stall is critical to determine if your workload on a runtime has an ITLB performance issue.

In Section B.6, we show that even while running in single instance mode, Ghost.js has a 6.47% stall due to ITLB misses. When large pages are implemented, the performance improves by 5% and the ITLB misses are reduced by 30% and the ITLB Miss Stall is reduced from 6.47% to 2.87% .

Another key metric is the ITLB Misses Per Kilo Instructions (MPKI). This metric is a normalization of the ITLB misses against number of instructions, and it allows comparison between different systems. This metric is calculated using two PMU counters: ITLB_MISSES.WALK_COMPLETED and INST_RETIRED.ANY, as described in Equation 2. There are distinct PMU counters for large pages and 4KB pages, so Equation 2 shows the calculation for each PMU counter, respectively.

Equation 2: Calculating ITLB MPKI

$$ITLB_MPKI = 1000 * (\frac{ITLB_MISSES.WALK_COMPLETED}{INST_RETIRED.ANY})$$

$$ITLB_4K_MPKI = 1000 * (\frac{ITLB_MISSES.WALK_COMPLETED_4K}{INST_RETIRED.ANY})$$

$$ITLB_2M_4M_MPKI = 1000 * (\frac{ITLB_MISSES.WALK_COMPLETED_2M_4M}{INST_RETIRED.ANY})$$

Upon calculating the MPKI for the runtime workloads in Figure B-1, we find that the ITLB MPKI and the ITLB 4K MPKI are very close to each other across the workloads. We can thus infer that most of the misses are from 4KB page walks. Another observation is that the benchmarks have lower ITLB MPKI than large real world software, which means that optimization decisions made on benchmarks might not translate to open source software.

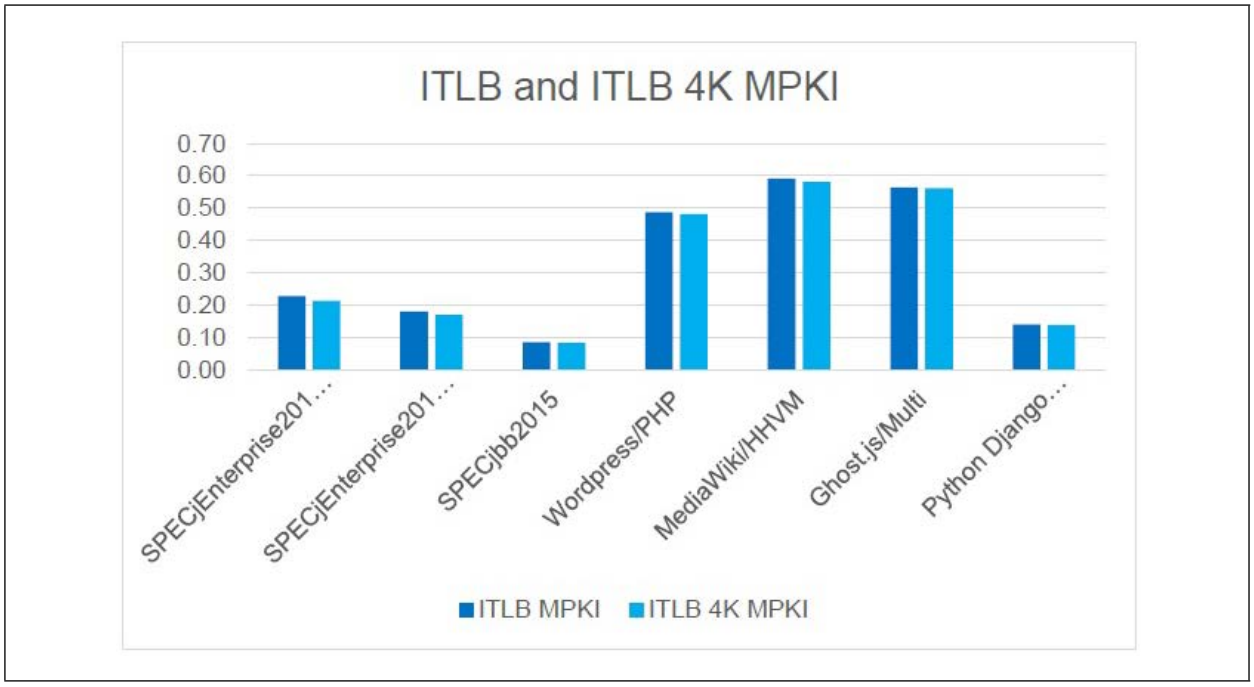


Figure B-1. ITLB and ITLB 4K MPKI Across Runtime Workloads

Having ITLB MPKI also enables us to do comparisons across different systems and workloads. Table B-3 compiles the ITLB MPKI across various workloads published[1,2]. We can observe that there is not a direct correlation of binary size to ITLB MPKI. Some smaller binaries, such as MySQL, have one of the largest ITLB MPKI. When multiple threads are active, the ITLB MPKI almost doubles for both Ghost.js (single instance vs. multi instance) and Clang (-j1 vs -j4). The

1. Ottoni, G., & Bertrand, M. (2017). Optimizing Function Placement for Large-Scale Data-Center Applications. CGO 2017.
2. Lavaee, R., Criswell, J., & Ding, C. (Oct 2018). Codestitcher: Inter-Procedural Basic Block Layout Optimization. arXiv:1810.00905v1 [cs.PL].

ITLB MPKI is much lower on newer servers (using Intel® Xeon® 8180 processors) as compared to older generation servers (using Intel® Xeon® E5 processors).

Table B-3. ITLB MPKI and Executable Sizes Across Various Workloads

Workload	Text (MB)	ITLB MPKI	System Details
AdIndexer HHVM Multifeed TAO	186 133 199 70	0.48 1.28 0.40 3.08	Dual 2.8 GHz Intel® Xeon® E5-2680 v2 (based on Ivy Bridge microarchitecture) server platform, with 10 cores and 25 MB LLC per processor.
MySQL Clang –j4 Clang –j1 Firefox Apache PHP (w opcache) Apache PHP (w/o opcache) Python	15 50 50 81 16 16 2	9.35 2.23 1.01 1.54 0.33 0.96 0.19	Two dual core Intel® Core™ i5-4278U (based on Haswell microarchitecture) processors running at 2.60 GHz. 32 KB instruction cache and a 256 KB second level unified cache private to each core. Both caches are 8-way set associative. The last level cache is 3 MB, 12-way set associative, and is shared by all core.
SPECjEnterprise2018-WP-VM SPECjEnterprise2018-WP-Native SPECjbb2015 Wordpress/PHP MediaWiki/HHVM Ghost.js/Multi Ghost.js/Single Python Django (Instagram)		0.23 0.18 0.09 0.49 0.59 0.56 0.23 0.14	Intel® Xeon® Platinum 8180 (based on Skylake Server microarchitecture) with 112 cores @ 2.5 GHz (except MediaWiki/HHVM which is on a SKX-D with 18 cores).

B.2.2 MEASURING THE ITLB MISS STALL

Intel has a number of tools to automate measuring ITLB miss stalls, including Intel® VTune™ Profiler, EMON/EDP, and Linux PMU tools. This blueprint offers a convenient tool (**measure-perf-metric.sh**) based on perf to collect and derive various stall metrics on Intel® Xeon® Scalable processors. The tool is open sourced and available for download at http://github.com/intel/iodlr. Figure B-2 shows the command line to collect and derive ITLB miss stalls for an application with process id=69772. The tool output shows the application has a 3.09% ITLB miss stall.

```
$ git clone http://github.com/intel/iodlr
$ export PATH=`pwd`/iodlr/tools/:$PATH
$ measure-perf-metric.sh -p 69772 -t 30 -m itlb_stalls
Initializing for metric: itlb_stalls
Collect perf data for 30 seconds
perf stat -e cycles,instructions,icache_64b.iftag_stall
-------------------------------------------------
Profile application with process id: 69772
-------------------------------------------------
Calculating metric for: itlb_stalls

=================================================
Final itlb_stalls metric
-------------------------------------------------
FORMULA: metric_ITLB_Misses(%) = 100*(a/b)
         where, a=icache_64b.iftag_stall
                b=cycles
=================================================
metric_ITLB_Misses(%)=3.09
```

Figure B-2. measure-perf-metric.sh Tool Usage for Process ID 69772 for 30 Seconds

Use the command "**measure-perf-metric.sh –h**" to display help messages for using the tool. Refer to the README.md file, which describes how to add new metrics to the tool.

B.2.3 SOURCE OF ITLB MISSES

The next task is to find where the ITLB misses are coming from. They could be coming from the .text segment of the runtime, JITted code, some other dynamic library of the runtime, or native libraries in the user code. Performance tools such as perf, are required to determine where the ITLB misses are coming from.

In the case of Ghost.js that we examined earlier, most of the ITLB misses are coming from the .text segment of the Node.js[1] binary. We find this to be the case for several other Node.js workloads. Using the current release of node.js (v12.8.0) and the **measure-pe**rf-metric.sh tool, we can determine it for a Node.js workload. Figure B-3 shows that 65.23% of the stalls are in the node binary. The '-r' option to measure-perf-metric.sh uses perf record underneath to record the location in the source code that is causing the **itlb_stalls**.

```
$ measure-perf-metric.sh -p 58448 -r -t 20 -m itlb_stalls
Samples: 77K of event 'icache_64b.iftag_stall', Event count (approx.): 1558
Overhead  Shared Object        Command
  65.23%  node                 node
  20.69%  perf-56817.map       node
   4.53%  libc-2.27.so         node
```

Figure B-3. Using measure-perf-metric.sh with -r to Determine Where TLB Misses are Coming From

While Figure B-3 shows where the TLB miss overheads are coming from in terms of stalled cycles, we further analyze the latest upstream node.js (14.0.0-pre) to find the overhead in terms of ITLB miss counts using the **perf record -e frontend_retired.tlb_miss**command. We extract the report using the `perf script` command and filtering it

1. NodeJS Foundation. (2019,. August). Node.js JavaScript Runtime. Retrieved from Node.js JavaScript Runtime: https://nodejs.org/en

based on the ITLB miss addresses. We find that 17.6% of ITLB misses are from JITted code and 72.8% from the node binary. We also find that "built-in" functions, which are part of node binary, account for 19.5% of the total ITLB misses.

On Linux and Windows systems, applications are loaded into memory into 4KB pages, which is the default on most systems. One way to reduce the ITLB misses is to use the larger page size, which has two benefits. The first benefit is fewer translations are required leading to fewer page walks. The second benefit is less space is used in the cache for storing translations, allowing more space to be available for the application code. Some older systems, such as one using Intel® Xeon® E5-2680 v2 processors (based on Ivy Bridge microarchitecture), have only 8 huge-page ITLB entries that are organized in a single level, so mapping all the text to large pages could cause a regression. However on Intel® Xeon® Platinum 8180 processors (based on Skylake Server microarchitecture), the STLB is shared by both 4KB and 2MB pages and has 1536 entries.

B.3 SOLUTION

B.3.1 LINUX* AND LARGE PAGES

On the Linux OS, there are two ways of using large pages in an application:

- **Explicit Huge Pages (hugetlbfs).** Part of the system memory is exposed as a file system that applications can mmap from. You can check the system through **cat /proc/meminfo** and see if lines like **HugePages_Total** are present.
- **Transparent Huge Pages (THP).** Linux also offers Transparent Hugepage Support which manages large pages automatically and is transparent for applications. The application can tell Linux to use large-pages-backed memory through madvise. You can check the system through **cat /sys/kernel/mm/transparent_hugepage/enabled.** If the values are **always** or **madvise**, then THP is available for the application. With `madvise`, THP is enabled only inside MADV_HUGEPAGE regions. Figure B-4 shows how to check the distribution for THP.

```
% cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
% cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```

Figure B-4. Commands for Checking Linux* Distribution for THP

B.3.2 LARGE PAGES FOR .TEXT

There are a few solutions on Linux for solving the ITLB miss issue for .text segments:

- **Linking runtime with libhugetlbfs:** There are a number of support utilities and a library packaged collectively as libhugetlbfs. The library provides support for automatically backing text, data, heap, and shared memory segments with huge pages. This relies on explicit huge pages that the system administrator has to manage using tools like hugeadm.
- **Using the Intel Reference Implementation:** The reference implementations have both a C and a C++ module that automates the process using Transparent Huge Pages. A couple of API calls described below may be invoked at the beginning of the runtime to map a subset of the application's .text segment to 2MB pages.

- **Using an explicit option or flag in the runtime:** The Node.js runtime has an implementation that is exposed using **--enable-largepages=on** when you run Node.js. The PHP runtime has a flag that can be added to the .ini file. For details, see: https://www.php.net/manual/en/opcache.configuration.php.

B.3.3 REFERENCE CODE

This blueprint offers a reference implementation that enables an application to utilize large pages for its execution. The open source reference implementation is available for download at http://github.com/intel/iodlr. Both a C and C++ implementation are provided.

The following is a high level description of the reference implementation and its APIs.

1. Find the .text region in memory.
    a. Examine the /proc/self/maps to determine the currently mapped .text region and obtain the start and end addresses.
    b. Modify the start address to point to the very beginning of the .text segment.
    c. Align the start and end addresses to large page boundaries.
2. Move the .text region to large pages.
    a. Map a temporary area and copy the original code there.
    b. Use mmap using the start address with MAP_FIXED so we get exactly the same virtual address.
    c. Use `madvise` with MADV_HUGE_PAGE to use anonymous 2MB pages.
    d. If successful, copy the code from the temporary area and unmap it.

There are five API calls provided in the reference implementation as shown in Figure B-5. Since the initial release, the ability to map DSOs has been added.

```
/* Performs a platorm-dependent check to determine whether it is possible to map
   to large pages and stores the result of the check in result. */
map_status IsLargePagesEnabled(bool* result);

/* Attempts to map an application's .text region to large pages.
If the region is not aligned to 2 MiB then the portion of the page that lies below
the first multiple of 2 MiB remains mapped to small pages. Likewhise, if the region
does not end at an address that is a multiple of 2 MiB, the remainder of the region
will remain mapped to small pages. The portion in-between will be mapped to large
pages. */
map_status MapStaticCodeToLargePages();

/* Retrieves an address range from the process' maps file associated with a DSO
   whose name matches lib_regex and attempts to map it to large pages */
map_status MapDSOToLargePages(const char* lib_regex);

/* Attempts to map the given address range to large pages. */
map_status MapStaticCodeRangeToLargePages(void* from, void* to);

/* A string containing the textual error message. The string is owned by the
implementation and must not be freed. */
const char* MapStatusStr(map_status status, bool fulltext);
```

Figure B-5. API Calls Provided by the Intel Reference Implementation

B.3.4 LARGE PAGES FOR THE HEAP

The Just-In-Time (JIT) compiler compiles methods on demand and the memory for the JITted code is allocated from the heap and subject to garbage collection.

The runtime can allocate heap on large pages using `mmap` with the flags argument set as **MAP_HUGETLB** (available since Linux 2.6.32) or **MAP_HUGE_2MB/ MAP_HUGE_1GB** (available since Linux 3.8). Alternatively, the heap region can be set to use transparent huge pages on Linux by using the `madvise` system call with **MADV_HUGE_PAGES**. When using `madvise`, the runtime must check that **transparent_hugepage** is set appropriately in the OS as either madvise or always, and not set to never.

The Java* VM has several options for mapping the Java heap with large pages[1]. Since the JITted code is also on the heap, it allocates both the code and the data to large pages.

-XX:+UseHugeTLBFS mmaps Java heap into hugetlbfs, which should be prepared separately.

-XX:+UseTransparentHugePages madvise-s that Java heap should use THP.

1. Aleksey Shipilev, Redhat. (2019, 03 03). Transparent Huge Pages. Retrieved from JVM Anatomy Quarks: https://shipilev.net/jvm/anatomy-quarks/2-transparent-huge-pages/.

B.4 SOLUTION INTEGRATION

Integrating the solution into a new runtime requires the following changes:

1. Follow the style guide of the runtime and update the reference code.
2. Determine in the runtime where to make the API calls to remap the **.text** segment.
3. Change the build to link with the new files/library.
4. Provide a build time or runtime option to turn on this feature.

B.4.1 V8 INTEGRATION WITH THE REFERENCE IMPLEMENTATION

V8[1] is the Google* open source high-performance JavaScript* engine, written in C++. We integrated Intel's large pages reference implementation with V8 using the steps described above.

Here are the specific steps that we used to integrate the reference implementation within V8:

1. Check out, configure, and build v8 from: https://v8.dev/docs/build-gn.
2. Add call to **MapStaticCodeToLargePages()** at the beginning of **Shell::Main()** in d8.cc. Include **huge_page.h** in the source file.
3. Generate build files with the command:

   gn gen out/foo –args='is_debug=false target_cpu="x64" is_clang=false'
4. Update the following build files:

   a. Update **out/foo/obj/d8.ninja**

   Add **–Ipath/to/huge_page.h** to include_dirs variable

   Add **-Wl,-T path/to/ld.implicit.script** to ldflags variable

   b. Update out/foo/toolchain.ninja

   Add **path/to/libhuge_page.a –lstdc++** to link command, before –Wl,–endgroup
5. Compile V8 with the command:

   ninja -C out/bar/ d8

B.4.2 JAVA JVM INTEGRATION WITH THE REFERENCE IMPLEMENTATION

OpenJDK is a free and open-source implementation of the Java Platform, Standard Edition written in a combination of C and C++. We determined that Java executable unlike v8 or nodejs is a thin 'C' wrapper that uses dlopen to load the libjvm.

We integrated Intel's C large pages reference implementation with OpenJDK. Here are the specific steps that we used to integrate the reference implementation:

1. Check out, configure, and build OpenJDK using instructions at: http://cr.openjdk.java.net/~ihse/demo-new-build-readme/common/doc/building.html
2. Modify the code in **src/java.base/unix/native/libjli/java_md_solinux.c** to load libjvm.so into 2M pages:
   - **Use the API to check if LargePages is supported.**
   - **Use the API `MapDSOToLargePages(const char* lib_regex)` to load libjvm.so into 2M pages.**
3. Compile and rebuild the Java wrapper.

1. Google V8 JavaScript. (2019, August). V8 JavaScript Engine. Retrieved from V8 JavaScript Engine: https://v8.dev/

B.5 LIMITATIONS

There are several limitations to be aware of when using large pages:

- Fragmentation is an issue that is introduced when using large pages. If there is insufficient contiguous memory to assemble the large page, the operating system tries to reorganize the memory to satisfy the large page request, which can result in longer latencies. This can be mitigated by allocating large pages explicitly ahead of time. The reference code does not have support for explicit huge pages.
- Another issue is the additional execution time it takes to perform the algorithm in the Intel reference code. For short running programs, it adds additional execution time and might result in a slowdown rather than a speedup.
- We have recently encountered an issue when the current implementation is used with multiple instances of the same application. We have a report that it increases the LLC misses. We think this is due to the kernel not sharing the code after the remapping. We are investigating and working on a solution.

Tools like perf are no longer able to follow the symbols after the .text is remapped (Figure B-6) and the perf output will not have the symbols. You will need to provide the static symbols to perf in **/tmp/perf-PID.map** at startup.

```
1.35%  node  perf-12142.map          [.] 0x0000562eb19e86aa
1.19%  node  perf-12142.map          [.] 0x0000562eb19e8803
0.71%  node perf-12142.map           [.] 0x0000562eb19e891f
```

Figure B-6. perf Output Will Not Have the Proper Symbols After Large Page Mapping

B.6 CASE STUDY

This section details how this optimization helps performance and reduces ITLB misses in three workloads in three environments. The workloads are:

- Ghost[1], a fully open source, adaptable platform for building and running a modern online publication.
- Web Tooling[2], a suite designed to measure JavaScript-related workloads.
- MediaWiki[3], a free and open-source wiki engine written in PHP.

This case study uses data running on the Intel® Xeon® Platinum 8180 processor (based on Skylake Server microarchitecture) for Ghost.js and Web Tooling and uses the Intel® Xeon® D-2100 processor[4] for MediaWiki to showcase the benefits of large pages. The last case study demonstrates how to use Visualization tools to identify patterns in the data.

---

1. Ghost Team. (n.d.). Ghost: The professional publishing platform. Retrieved from Ghost Non Profit Web Site: https://ghost.org/.
2. Google Web Tooling. (2019, August). Web Tooling Benchmark. Retrieved from Web Tooling Benchmark: https://github.com/v8/web-tooling-benchmark.
3. WikiMedia Foundation. (2019, August). Mediawiki Software. Retrieved from Mediawiki Software: http://mediawiki.org/wiki/MediaWiki.
4. Xeon-D, I. (n.d.). Xeon-D. Retrieved from intel.com: https://www.intel.com/content/www/us/en/products/processors/xeon/d-processors.html.

B.6.1 GHOST.JS WORKLOAD

Ghost is an open source blogging platform written in JavaScript and running on Node.js. We created a workload that has a single instance Ghost.js running on Node.js as a server and uses Apache Bench as a client to make requests. The performance is measured by a metric called Requests Per Second (RPS).

Intel developed and contributed the 2MB for Code PR which is now merged into Node.js master. You can turn on large pages at runtime with the switch `--enable-largepages=on` in recent builds of Node.js. In older builds, you can enable large pages by building with `--use-largepages`. You can then compare the default build of Node.js with a build configured to use large pages.

Table B-4 shows the key metrics with Node.js and Node.js with large pages. RPS improves by 5%. The stalls due to the ITLB misses have reduced by 56%. The ITLB MPKI improved by 57%. The gains are mostly coming from the ITLB walk reduction which reduces by 55%. Note that the number of 2MB walks increases due to the use of 2MB pages.

Table B-4. Key Metrics for Ghost.js With and Without Large Pages

Metric	Node.js With Large Pages	Node.js Without Large Pages (Default)	Large Pages/Default
Throughput: Requests Per Second (RPS)	134.32	127.23	1.05
metric_ITLB_Misses(%)	2.87	6.47	0.44
metric cycles per txn	30048182	31426008	0.95
metric instructions per txn	46703106	47153431	0.99
ITLB_MISSES.WALK_COMPLETED	36799580	82504511	0.45
ITLB_MISSES.WALK_COMPLETED_2M_4M	938969	250959	3.74
ITLB_MISSES.WALK_COMPLETED_4K	35842004	82166894	0.44
metric ITLB MPKI	0.098	0.230	0.43
metric ITLB 4K MPKI	0.096	0.229	0.43
metric ITLB 2M_4M MPKI	0.002	0.0007	3.55

B.6.2 WEB TOOLING WORKLOAD

B.6.2.1 Node Version

Clear Linux* OS distributes Node 10.16.0 compiled with large pages support. We installed official Node 10.16.0 (which does not have large page support) on Ubuntu* 18.04 so we can compare the same version. Ubuntu only comes with Node 8.10.0 as part of the apt repository.

B.6.2.2 Web Tooling

This is a suite designed to measure the JavaScript-related workloads commonly used by web developers, such as the core workloads in popular tools like Babel or TypeScript. It has a number of sub-components and reports a throughput score.

B.6.2.3 Comparing Clear Linux* OS and Ubuntu*

Table B-5 shows the key metrics when running the Web Tooling workload. Although we observed small improvements in the throughput and cycles on Clear Linux when compared to Ubuntu, the Clear Linux reduces the ITLB miss stall by 59%, the ITLB MPKI by 51%, and the 4KB MPKI by 52%. This is due to Clear Linux distributing Node.js compiled with the `--use-largepages` option. The throughput isn't impacted significantly, since the ITLB stalls were not as significant to begin with.

Table B-5. Key Metrics for Web Tooling across Clear Linux and Ubuntu 18.04

Metric	Clear Linux	Ubuntu 18.04	Clear Linux/Ubuntu
Throughput	10.91	10.80	1.01
metric_ITLB_Misses(%)	0.91	2.21	0.41
metric cycles per txn	531,821,553.08	537,631,089.06	0.98
metric instructions per txn	836,955,459.53	879,834,649.97	0.95
ITLB_MISSES.WALK_COMPLETED	8,145,008	17,230,161	0.47
ITLB_MISSES.WALK_COMPLETED_4K	7,909,985	17,070,769	0.46
ITLB_MISSES.WALK_COMPLETED_2M_4M	241,215	142,356	1.69
metric ITLB MPKI	0.0298	0.0604	0.49
metric ITLB 4K MPKI	0.0289	0.0598	0.48
metric ITLB 2M_4M MPKI	0.0009	0.0005	1.76

B.6.3 MEDIAWIKI WORKLOAD

MediaWiki is a free and open-source wiki engine written in PHP. We used HHVM 3.25 to execute MediaWiki. HHVM maps the hot text pages to 2MB pages[1] and uses both the 4 KB and 2 MB pages. HHVM provides command line options `-vEval.MaxHotTextHugePages` and `-vEval.MapTCHuge` to enable large pages for the hot text pages and the Translation Cache pages (which holds the JIT generated code). In addition, it relies on code ordering to reduce the TLB misses. Table B-5 shows the improvement in metrics with large pages. There is a reduction of 16% for the ITLB miss stalls, a 29% reduction for the overall walks completed, and a 66% reduction in the hits to the shared TLBs. Skylake Server microarchitecture introduced new precise front-end events (e.g.,

1. Ottoni, G., & Bertrand, M. (2017). Optimizing Function Placement for Large-Scale Data-Center Applications. CGO 2017.

FRONTEND_RETIRED.ITLB_MISS counts retired instructions that experienced ITLB (Instruction TLB) true miss) and we can see that all those are lower with large pages with STLB_MISS reducing by 23%.

Table B-6. Key Metrics for MediaWiki Workload on HHVM

Metric	Large Pages	No Large Pages	Large/No Large
metric_ITLB_Misses(%)	2.86	3.42	0.84
metric cycles per txn	44339066	44372158	0.99
metric instructions per txn	39129166	39168226	0.99
ITLB_MISSES.WALK_COMPLETED	7735.06	10850.57	0.71
ITLB_MISSES.WALK_COMPLETED_2M_4M	281.39	243.89	1.15
ITLB_MISSES.WALK_COMPLETED_4K	7453.67	10606.68	0.70
FRONTEND_RETIRED.ITLB_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L1I_MISS	160403.87	162401.23	0.99
FRONTEND_RETIRED.L2_MISS	19566.98	20075.91	0.97
FRONTEND_RETIRED.STLB_MISS	3820.17	4961.98	0.77
metric ITLB MPKI	0.2426	0.351	0.69
metric ITLB 4K MPKI	0.2310	0.341	0.67
metric ITLB 2M_4M MPKI	0.0116	0.009	1.18

B.6.4 VISUALIZATION OF BENEFITS

B.6.4.1 Precise Events

Intel PMU has Precise Event-Based Sampling (PEBS) which reports precise information, such as instruction address of cache misses. On the Intel® Xeon® Platinum 8180 (based on Skylake Server microarchitecture), the PEBS events support additional front-end events which are hardest to locate in the source code. Two of them are for ITLB misses as shown in Table B-7.

Table B-7. Precise Front-end Events for ITLB Misses

Event	Description
FRONTEND_RETIRED.ITLB_MISS (1st level)	Retired instructions following a true ITLB miss.
FRONTEND_RETIRED.STLB_MISS (2nd level)	Retired instructions following an ITLB and STLB miss (2nd level).

B.6.4.2 Visualizing Precise ITLB Miss

We can use Linux `perf` to record the frontend_retired.itlb_miss event and visualize the events with FlameScope, an Open Source tool from Netflix*. FlameScope is a visualization tool for exploring time ranges heatmaps and FlameGraphs. FlameScope starts by visualizing the input data as an interactive subsecond-offset heat map.

```
$ perf record -o /tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out.b -b  -e
frontend_retired.itlb_miss -- /home/dslo/ssuresh1/node-v12.14.1/node.orig --perf-
basic-prof dist/cli.js
[ perf record: Woken up 8 times to write data ]
[ perf record: Captured and wrote 1.944 MB
/tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out.b (1539 samples) ]

$ perf script -i /tmp/webtooling.node12.14.1.callgraph.itlb_miss.orig.out --header >
webtooling.node12.14.1.itlb_miss.orig
```

Figure B-7. Using Perf Record with -e frontend_retired.itlb_miss to Determine ITLB Misses

The output of perf script can be imported into FlameScope and we can visualize the ITLB misses. We can see that some portions of the workload have much more ITLB misses than others. When we compare Figure B-8 and Figure B-9 we can see that the heatmap is much sparser for the ITLB misses when we are using large pages in Node.js.

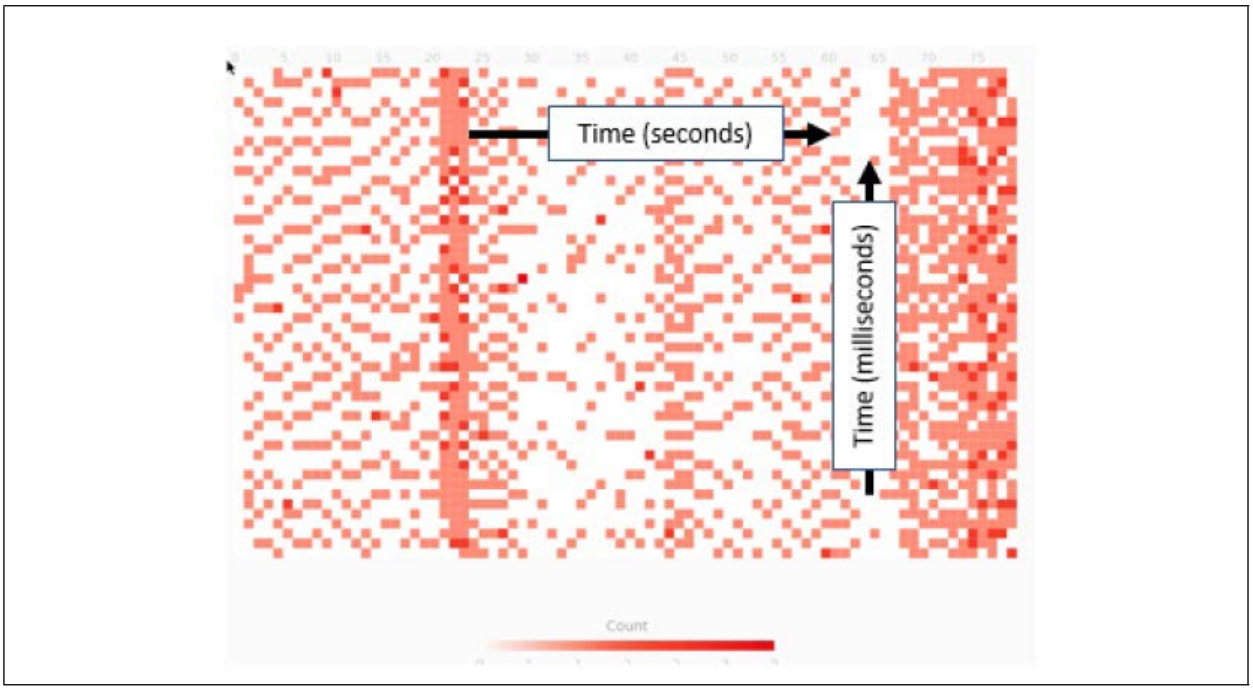


Figure B-8. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload

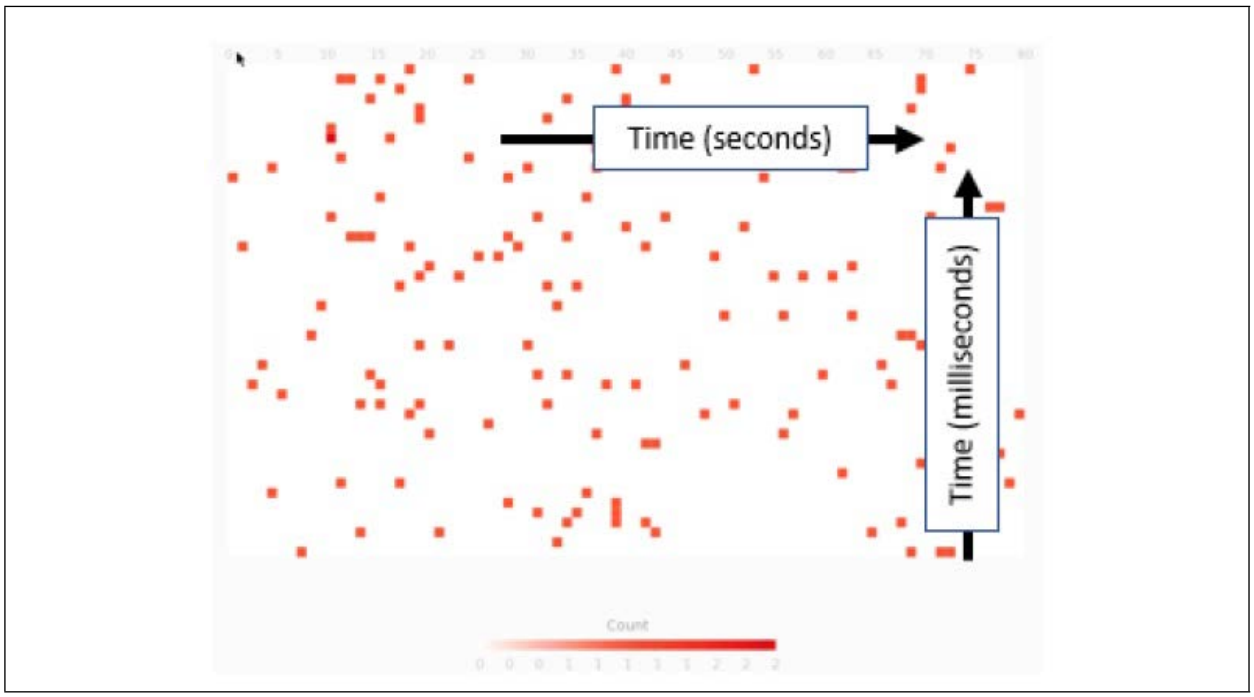


Figure B-9. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload

We also visualize the ITLB miss counts for the v8 "Built-in" functions by extracting the ITLB misses associated with the "Built-in" function from the `perf script` output. We plot the graph with the virtual address of the "Built-in" functions on y-axis and time on the x-axis (Figure B-10 and Figure B-11). Similar to FlameScope graphs, ITLB misses are sparser when we are using large pages in Node.js.

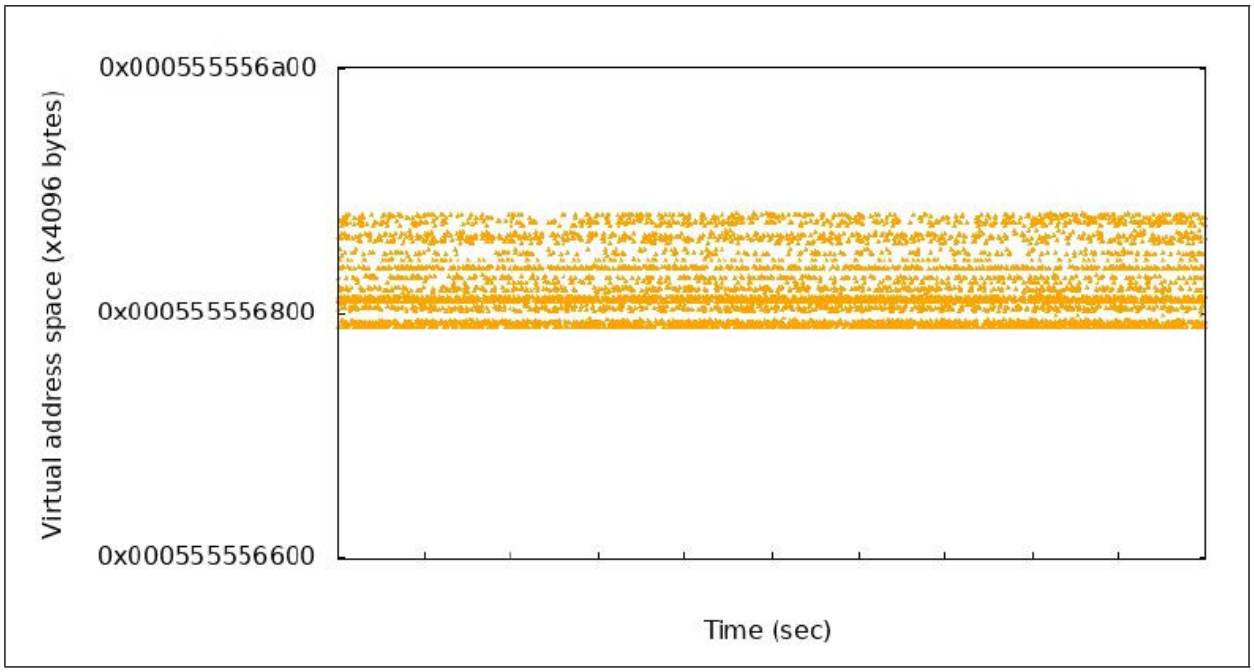


Figure B-10. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload

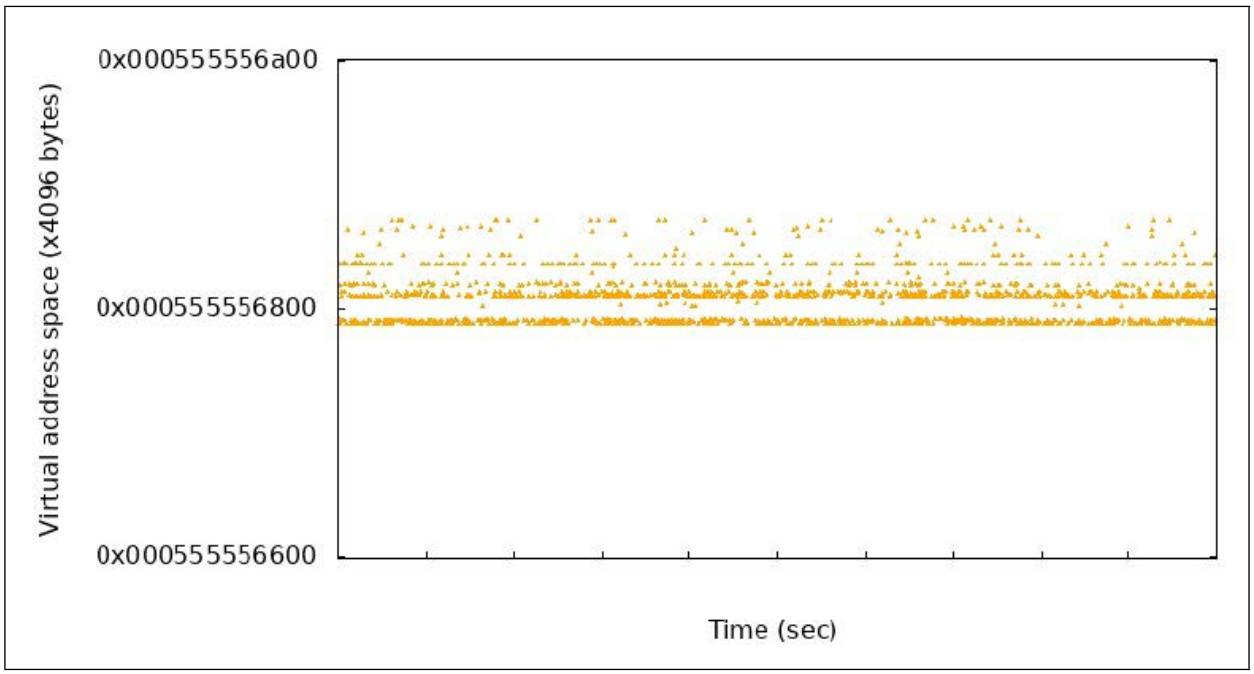


Figure B-11. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload When Run With Large Pages

B.7 SUMMARY

This runtime optimization blueprint described the problem that runtimes have with high ITLB miss stalls, and discussed how to diagnose the problem, as well as techniques and a reference implementation to solve the problem. A case study showed the benefits of integrating the solution into a new runtime. The three examples in the case study demonstrated that the use of 2M pages has the potential to improve ITLB Miss Stalls by 43%, ITLB Walks by 45%, and ITLB MPKI by 46%.

B.8 TEST CONFIGURATION DETAILS

Test configuration details are provided in the tables below.

Table B-8. System Details

System Info	DSLOHost011
Manufacturer	Intel Corporation
Product Name	S2600WFT
BIOS Version	SE5C620.86B.0X.01.0115.012820180604
OS	Ubuntu 18.04.3 LTS
Kernel	4.15.0-58-generic

Table B-8. System Details (Contd.)

System Info	DSLOHost011
Microcode	0x200005e

Table B-9. Processor Information

Model Name	Intel® Xeon® Platinum 8180 CPU @ 2.50GHz
Sockets	2
Hyper-Threading Enabled	Yes
Total CPU(s)	112
NUMA Nodes	2
NUMA cpulist	0-27,56-83 :: 28-55,84-111
L1d Cache	32K
L1i Cache	32K
L2 Cache	1024K
L3 Cache	39424K
Prefetchers Enabled	DCU HW, DCU IP, L2 HW, L2 Adj.
Turbo Enabled	True
Power & Perf Policy	Balanced
CPU Freq Driver	intel_pstate
CPU Freq Governor	powersave
Current CPU Freq MHz	1000
AVX2 Available	True
AVX512 Available	True
AVX512 Test	Passed
PPIN (CPU0)	c6aa1d2bcbba4d86

Table B-10. Kernel Vulnerability Status

Vulnerabilities	**DSLOHost011**
CVE-2017-5753	OK (Mitigation: usercopy/swapgs barriers and __user pointer sanitization)
CVE-2017-5715	OK (Full retpoline + IBPB are mitigating the vulnerability)

Table B-10. Kernel Vulnerability Status (Contd.)

CVE-2017-5754	OK (Mitigation: PTI)
CVE-2018-3640	OK (your CPU microcode mitigates the vulnerability)
CVE-2018-3639	OK (Mitigation: Speculative Store Bypass disabled via prctl and seccomp)
CVE-2018-3615	OK (your CPU vendor reported your CPU model as not vulnerable)
CVE-2018-3620	OK (Mitigation: PTE Inversion)
CVE-2018-3646	OK (this system is not running a hypervisor)
CVE-2018-12126	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2018-12130	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2018-12127	OK (Mitigation: Clear CPU buffers; SMT vulnerable)
CVE-2019-11091	OK (Mitigation: Clear CPU buffers; SMT vulnerable)

B.9 ADDITIONAL REFERENCES

In addition to the references cited in this appendix, the following references were used:

Ahmad Yasin, Intel Corporation. (2014). A Top-Down method for performance analysis and counters architecture. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS , (pp. 35-44).

Performance Monitoring Event List. Retrieved from 01.org: https://download.01.org/perfmon/SKX/.

Panchenko, M. (2017). Building Binary Optimizer with LLVM. Retrieved from LLVM.ORG: https://llvm.org/devmtg/2016-03/Presentations/BOLT_EuroLLVM_2016.pdf.

M. Panchenko, R. Auler, B. Nell and G. Ottoni, "BOLT: A Practical Binary Optimizer for Data Centers and Beyond," 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, 2019, pp. 2-14, doi: 10.1109/CGO.2019.8661201..

CHAPTER 1
INTRODUCTION

1.1 ABOUT THIS MANUAL 1-1
1.2 TUNING YOUR APPLICATION 1-1
1.3 INTEL PROCESSORS SUPPORTING THE INTEL® 64 ARCHITECTURE 1-2
1.4 ORGANIZATION OF THIS MANUAL 1-4
1.4.1 Chapter Summaries 1-4
1.4.1.1 Volume 1 1-4
1.4.1.2 Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures 1-5
1.5 GLOSSARY 1-6
1.6 RELATED INFORMATION 1-7

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

2.1 6TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY 2-1
2.1.1 The Redwood Cove Microarchitecture 2-1
2.1.1.1 Branch Hint 2-2
2.1.1.2 Profiling Support for Branch Hints 2-3
2.1.1.3 New Redwood Cove Macro-Fusions 2-3
2.1.1.4 Improved Memory BW 2-4
2.1.1.5 Array of Pointers Prefetcher 2-4
2.1.1.6 LLC Page Prefetcher (LLCPP) 2-5
2.2 THE SAPPHIRE RAPIDS MICROARCHITECTURE 2-5
2.2.1 4th Generation Intel® Xeon® Scalable Family of Processors 2-6
2.3 THE ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE 2-6
2.3.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture 2-6
2.3.2 Hybrid Scheduling 2-6
2.3.2.1 Intel® Thread Director 2-6
2.3.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture 2-9
2.3.2.3 Scheduling with a Multi-E-Core Module 2-10
2.3.2.4 Scheduling Background Threads on x86 Hybrid Architecture 2-10
2.3.3 Recommendations for Application Developers 2-10
2.4 THE GOLDEN COVE MICROARCHITECTURE 2-10
2.4.1 Golden Cove Microarchitecture Overview 2-11
2.4.1.1 Cache Subsystem and Memory Subsystem 2-15
2.4.1.2 Avoiding Destination False Dependency 2-15
2.5 ICE LAKE CLIENT MICROARCHITECTURE 2-16
2.5.1 Ice Lake Client Microarchitecture Overview 2-16
2.5.1.1 The Front End 2-18
2.5.1.2 The Out of Order and Execution Engines 2-18
2.5.1.3 Cache and Memory Subsystem 2-20
2.5.1.4 Fast Store Forwarding Prediction (FSFP) 2-23
2.5.1.5 New Instructions 2-24
2.5.1.6 Ice Lake Client Microarchitecture Power Management 2-25
2.6 SKYLAKE SERVER MICROARCHITECTURE 2-25
2.6.1 Skylake Server Microarchitecture Cache 2-27
2.6.1.1 Larger Mid-Level Cache 2-27
2.6.1.2 Non-Inclusive Last Level Cache 2-27
2.6.1.3 Skylake Server Microarchitecture Cache Recommendations 2-27
2.6.2 Non-Temporal Stores on Skylake Server Microarchitecture 2-29
2.6.3 Skylake Server Power Management 2-29
2.7 SKYLAKE CLIENT MICROARCHITECTURE 2-33
2.7.1 The Front End 2-34
2.7.2 The Out-of-Order Execution Engine 2-34
2.7.3 Cache and Memory Subsystem 2-36
2.7.4 Pause Latency in Skylake Client Microarchitecture 2-37

2.8 INTEL® 64 AND IA-32 INSTRUCTION BEST PRACTICES 2-39
2.8.1 Non-Privileged Instruction Serialization 2-39
2.8.2 Intel® Hyper-Threading Technology (Intel® HT Technology) 2-40
2.8.2.1 Processor Resources and Intel® HT Technology 2-41
2.8.2.2 Microarchitecture Pipeline and Intel® HT Technology 2-42
2.8.2.3 Execution Core 2-42
2.8.2.4 Retirement 2-42
2.8.3 SIMD Technology 2-42
2.8.4 Summary of SIMD Technologies and Application Level Extensions 2-44
2.8.4.1 MMX™ Technology 2-45
2.8.4.2 Streaming SIMD Extensions 2-45
2.8.4.3 Streaming SIMD Extensions 2 2-45
2.8.4.4 Streaming SIMD Extensions 3 2-45
2.8.4.5 Supplemental Streaming SIMD Extensions 3 2-45
2.8.5 SSE4.1 2-46
2.8.5.1 SSE4.2 2-46
2.8.5.2 AESNI and PCLMULQDQ 2-46
2.8.5.3 Intel® Advanced Vector Extensions (Intel® AVX) 2-47
2.8.5.4 Half-Precision Floating-Point Conversion (F16C) 2-47
2.8.5.5 RDRAND 2-47
2.8.5.6 Fused-Multiply-ADD (FMA) Extensions 2-47
2.8.5.7 Intel® Advanced Vector Extensions 2 (Intel® AVX2) 2-48
2.8.5.8 General-Purpose Bit-Processing Instructions 2-48
2.8.5.9 RDSEED 2-48
2.8.5.10 ADCX and ADOX Instructions 2-48

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

3.1 PERFORMANCE TOOLS 3-1
3.1.1 Intel® C++ and Fortran Compilers 3-1
3.1.2 General Compiler Recommendations 3-2
3.1.3 VTune™ Performance Analyzer 3-2
3.2 PROCESSOR PERSPECTIVES 3-2
3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy 3-2
3.2.2 Transparent Cache-Parameter Strategy 3-3
3.2.3 Threading Strategy and Hardware Multithreading Support 3-3
3.3 CODING RULES, SUGGESTIONS, AND TUNING HINTS 3-3
3.4 OPTIMIZING THE FRONT END 3-4
3.4.1 Branch Prediction Optimization 3-4
3.4.1.1 Eliminating Branches 3-4
3.4.1.2 Static Prediction 3-6
3.4.1.3 Inlining, Calls, and Returns 3-7
3.4.1.4 Code Alignment 3-8
3.4.1.5 Branch Type Selection 3-8
3.4.1.6 Loop Unrolling 3-10
3.4.2 Fetch and Decode Optimization 3-11
3.4.2.1 Optimizing for Microfusion 3-11
3.4.2.2 Optimizing for Macrofusion 3-12
3.4.2.3 Length-Changing Prefixes (LCP) 3-16
3.4.2.4 Optimizing the Loop Stream Detector (LSD) 3-18
3.4.2.5 Optimization for Decoded ICache 3-18
3.4.2.6 Other Decoding Guidelines 3-20
3.5 OPTIMIZING THE EXECUTION CORE 3-20
3.5.1 Instruction Selection 3-20
3.5.1.1 Integer Divide 3-21
3.5.1.2 Using LEA 3-21
3.5.1.3 ADC and SBB in Sandy Bridge Microarchitecture 3-23

3.5.1.4 Bitwise Rotation 3-24
3.5.1.5 Variable Bit Count Rotation and Shift. 3-24
3.5.1.6 Address Calculations 3-24
3.5.1.7 Clearing Registers and Dependency Breaking Idioms 3-25
3.5.1.8 Compares 3-27
3.5.1.9 Using NOPs 3-27
3.5.1.10 Mixing SIMD Data Types 3-28
3.5.1.11 Spill Scheduling. 3-28
3.5.1.12 Zero-Latency MOV Instructions 3-29
3.5.2 Avoiding Stalls in Execution Core 3-31
3.5.2.1 Writeback Bus Conflicts 3-31
3.5.2.2 Bypass Between Execution Domains 3-31
3.5.2.3 Partial Register Stalls 3-32
3.5.2.4 Partial XMM Register Stalls 3-32
3.5.2.5 Partial Flag Register Stalls 3-33
3.5.2.6 Floating-Point/SIMD Operands 3-34
3.5.3 Vectorization 3-35
3.5.4 Optimization of Partially Vectorizable Code 3-35
3.5.4.1 Alternate Packing Techniques. 3-37
3.5.4.2 Simplifying Result Passing 3-37
3.5.4.3 Stack Optimization 3-38
3.5.4.4 Tuning Considerations 3-39
3.6 OPTIMIZING MEMORY ACCESSES 3-40
3.6.1 Load and Store Execution Bandwidth 3-41
3.6.1.1 Making Use of Load Bandwidth in Sandy Bridge Microarchitecture 3-41
3.6.1.2 L1D Cache Latency in Sandy Bridge Microarchitecture. 3-42
3.6.1.3 Handling L1D Cache Bank Conflict 3-43
3.6.2 Minimize Register Spills 3-44
3.6.3 Enhance Speculative Execution and Memory Disambiguation 3-45
3.6.4 Store Forwarding. 3-45
3.6.4.1 Store-to-Load-Forwarding Restriction on Size and Alignment 3-46
3.6.4.2 Store-Forwarding Restriction on Data Availability 3-49
3.6.5 Data Layout Optimizations 3-50
3.6.6 Stack Alignment. 3-52
3.6.7 Capacity Limits and Aliasing in Caches 3-53
3.6.8 Mixing Code and Data. 3-53
3.6.8.1 Self-Modifying Code (SMC) 3-54
3.6.8.2 Position Independent Code 3-54
3.6.9 Write Combining 3-54
3.6.10 Locality Enhancement. 3-55
3.6.11 Non-Temporal Store Bus Traffic. 3-56
3.7 PREFETCHING 3-57
3.7.1 Hardware Instruction Fetching and Software Prefetching 3-57
3.7.2 Hardware Prefetching for First-Level Data Cache 3-58
3.7.3 Hardware Prefetching for Second-Level Cache 3-60
3.7.4 Cacheability Instructions 3-60
3.7.5 REP Prefix and Data Movement 3-61
3.7.6 Enhanced REP MOVSB and STOSB Operation. 3-63
3.7.6.1 Fast Short REP MOVSB. 3-63
3.7.6.2 Memcpy Considerations 3-63
3.7.6.3 Memmove Considerations 3-65
3.7.6.4 Memset Considerations. 3-65
3.8 REP STRING OPERATIONS 3-66
3.8.1 Fast Zero Length REP MOVSB. 3-66
3.8.2 Fast Short REP STOSB 3-66
3.8.3 Fast Short REP CMPSB and SCASB 3-66
3.9 FLOATING-POINT CONSIDERATIONS 3-66
3.9.1 Guidelines for Optimizing Floating-Point Code. 3-67
3.9.2 Floating-Point Modes and Exceptions 3-67

3.9.2.1 Floating-Point Exceptions 3-67
3.9.2.2 Dealing with Floating-Point Exceptions in x87 FPU Code 3-68
3.9.2.3 Floating-Point Exceptions in SSE/SSE2/SSE3 Code 3-68
3.9.3 Floating-Point Modes 3-69
3.9.3.1 Rounding Mode 3-69
3.9.3.2 Precision 3-71
3.9.4 x87 vs. Scalar SIMD Floating-Point Trade-Offs 3-71
3.9.4.1 Scalar Intel® SSE/Intel® SSE2 3-72
3.9.4.2 Transcendental Functions 3-72
3.10 MAXIMIZING PCIE PERFORMANCE 3-72
3.10.1 Optimizing PCIe Performance for Accesses Toward Coherent Memory and MMIO Regions (P2P) 3-73
3.11 SCALABILITY WITH CONTENDED LINE ACCESS IN 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS 3-73
3.11.1 Causes of Performance Bottlenecks 3-73
3.11.2 Performance Bottleneck Detection 3-73
3.11.3 Solutions for Performance Bottlenecks 3-75
3.11.4 Case Study: SysBench/MariaDB 3-76
3.11.5 Scalability With False Sharing 3-77
3.11.5.1 Causes of False Sharing 3-77
3.11.5.2 Detecting False Sharing 3-77
3.11.5.3 Fixing False Sharing and Additional Resources 3-78
3.11.5.4 Case Study: DeathStarBench/hotelReservation 3-79
3.11.6 Instruction Sequence Slowdowns 3-80
3.11.6.1 Causes of Instruction Sequence Slowdowns 3-80
3.11.6.2 Detecting Instruction Sequence Slowdowns 3-81
3.11.6.3 Fixing Instruction Sequence Slowdowns 3-81
3.11.7 Misprediction for Branches >2GB 3-82
3.11.7.1 Causes of Branch Misprediction >2GB 3-82
3.11.7.2 Detecting Branch Mispredictions >2GB 3-82
3.11.7.3 Fixing Branch Mispredictions >2GB 3-83
3.12 OPTIMIZING COMMUNICATION WITH PCI DEVICES ON INTEL® 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS 3-84
3.12.1 Signaling Devices with Direct Move 3-84
3.12.1.1 MOVDIR64B: Additional Considerations 3-85
3.12.1.2 Streaming Data 3-85
3.13 SYNCHRONIZATION 3-85
3.13.1 User-Level Monitor, User-Level MWAIT, and TPAUSE 3-85
3.13.1.1 Checking for User-Level Monitor, MWAIT, and TPAUSE Support 3-86
3.13.1.2 User-Level Monitor, User-Level MWAIT, and TPAUSE Operations 3-86
3.13.1.3 Recommended Usage of Monitor, MWAIT, and TPAUSE Operations 3-86

CHAPTER 4
INTEL ATOM® PROCESSOR ARCHITECTURES

4.1 THE CRESTMONT MICROARCHITECTURE 4-1
4.1.1 Crestmont Microarchitecture Overview 4-2
4.1.2 Predict and Fetch 4-3
4.1.3 Dynamic Load Balancing 4-4
4.1.4 Instruction Decode and the On-Demand Instruction Length Decoder (OD-ILD) 4-4
4.1.5 Allocation and Retirement 4-5
4.1.6 The Out-of-Order and Execution Engines 4-5
4.1.7 Cache and Memory Subsystem 4-6
4.1.8 Crestmont New Instruction Support 4-8
4.1.8.1 AVX-NE-CONVERT Instructions 4-8
4.1.8.2 AVX-IFMA 4-8
4.1.8.3 AVX-VNNI-INT8 Instructions 4-8
4.1.9 Legacy Intel® AVX1/Intel® AVX2 Instruction Support 4-8

4.1.9.1 256-bit Permute Operations 4-8
4.1.9.2 256-bit Broadcast with 128-bit Memory Operand 4-9
4.1.9.3 256-bit Insertion, Up-Conversion Instructions with 128-bit Memory Operand 4-9
4.1.9.4 256-bit Variable Blend Instructions 4-9
4.1.9.5 256-bit Vector TEST Instructions 4-10
4.1.9.6 The GATHER Instruction 4-10
4.1.9.7 Masked Load and Store Instructions 4-10
4.1.9.8 ADX Instructions 4-10
4.2 THE TREMONT MICROARCHITECTURE 4-10
4.2.1 Tremont Microarchitecture Overview 4-12
4.2.2 The Front End 4-12
4.2.3 The Out-of-Order and Execution Engines 4-13
4.2.4 Cache and Memory Subsystem 4-14
4.2.5 New Instructions 4-15
4.2.6 Tremont Microarchitecture Power Management 4-15

CHAPTER 5
CODING FOR SIMD ARCHITECTURES

5.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES 5-1
5.1.1 Checking for MMX Technology Support 5-2
5.1.2 Checking for Intel® Streaming SIMD Extensions (Intel® SSE) Support 5-2
5.1.3 Checking for Intel® Streaming SIMD Extensions 2 (Intel® SSE2) Support 5-2
5.1.4 Checking for Intel® Streaming SIMD Extensions 3 (Intel® SSE3) Support 5-3
5.1.5 Checking for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE) Support 5-3
5.1.6 Checking for Intel® SSE4.1 Support 5-4
5.1.7 Checking for Intel® SSE4.2 Support 5-4
5.1.8 Detection of PCLMULQDQ and AESNI Instructions 5-5
5.1.9 Detection of Intel® AVX Instructions 5-5
5.1.10 Detection of VEX-Encoded AES and VPCLMULQDQ 5-7
5.1.11 Detection of F16C Instructions 5-8
5.1.12 Detection of FMA 5-9
5.1.13 Detection of Intel® AVX2 5-10
5.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING 5-11
5.2.1 Identifying Hot Spots 5-13
5.2.2 Determine If Code Benefits by Conversion to SIMD Execution 5-13
5.3 CODING TECHNIQUES 5-13
5.3.1 Coding Methodologies 5-14
5.3.1.1 Assembly 5-15
5.3.1.2 Intrinsics 5-15
5.3.1.3 Classes 5-16
5.3.1.4 Automatic Vectorization 5-17
5.4 STACK AND DATA ALIGNMENT 5-18
5.4.1 Alignment and Contiguity of Data Access Patterns 5-18
5.4.1.1 Using Padding to Align Data 5-18
5.4.1.2 Using Arrays to Make Data Contiguous 5-18
5.4.2 Stack Alignment for 128-bit SIMD Technologies 5-19
5.4.3 Data Alignment for MMX™ Technology 5-20
5.4.4 Data Alignment for 128-bit data 5-20
5.4.4.1 Compiler-Supported Alignment 5-20
5.5 IMPROVING MEMORY UTILIZATION 5-22
5.5.1 Data Structure Layout 5-22
5.5.2 Strip-Mining 5-25
5.5.3 Loop Blocking 5-26
5.6 INSTRUCTION SELECTION 5-28
5.7 TUNING THE FINAL APPLICATION 5-30

CHAPTER 6
OPTIMIZING FOR SIMD INTEGER APPLICATIONS

6.1 GENERAL RULES ON SIMD INTEGER CODE 6-1
6.2 USING SIMD INTEGER WITH X87 FLOATING-POINT 6-2
6.2.1 Using the EMMS Instruction. 6-2
6.2.2 Guidelines for Using EMMS Instruction 6-2
6.3 DATA ALIGNMENT 6-3
6.4 DATA MOVEMENT CODING TECHNIQUES 6-5
6.4.1 Unsigned Unpack. 6-5
6.4.2 Signed Unpack 6-6
6.4.3 Interleaved Pack with Saturation. 6-6
6.4.4 Interleaved Pack without Saturation 6-8
6.4.5 Non-Interleaved Unpack. 6-8
6.4.6 Extract Data Element 6-10
6.4.7 Insert Data Element 6-10
6.4.8 Non-Unit Stride Data Movement. 6-12
6.4.9 Move Byte Mask to Integer 6-12
6.4.10 Packed Shuffle Word for 64-bit Registers 6-13
6.4.11 Packed Shuffle Word for 128-bit Registers 6-14
6.4.12 Shuffle Bytes 6-14
6.4.13 Conditional Data Movement 6-14
6.4.14 Unpacking/Interleaving 64-bit Data in 128-bit Registers. 6-15
6.4.15 Data Movement 6-15
6.4.16 Conversion Instructions 6-15
6.5 GENERATING CONSTANTS. 6-15
6.6 BUILDING BLOCKS 6-16
6.6.1 Absolute Difference of Unsigned Numbers 6-16
6.6.2 Absolute Difference of Signed Numbers 6-17
6.6.3 Absolute Value 6-17
6.6.4 Pixel Format Conversion. 6-17
6.6.5 Endian Conversion. 6-19
6.6.6 Clipping to an Arbitrary Range [High, Low] 6-20
6.6.6.1 Highly Efficient Clipping 6-21
6.6.6.2 Clipping to an Arbitrary Unsigned Range [High, Low] 6-22
6.6.7 Packed Max/Min of Byte, Word and Dword 6-22
6.6.8 Packed Multiply Integers 6-22
6.6.9 Packed Sum of Absolute Differences. 6-23
6.6.10 MPSADBW and PHMINPOSUW 6-23
6.6.11 Packed Average (Byte/Word). 6-23
6.6.12 Complex Multiply by a Constant 6-24
6.6.13 Packed 64-bit Add/Subtract 6-24
6.6.14 128-bit Shifts 6-24
6.6.15 PTEST and Conditional Branch 6-25
6.6.16 Vectorization of Heterogeneous Computations across Loop Iterations. 6-25
6.6.17 Vectorization of Control Flows in Nested Loops. 6-26
6.7 MEMORY OPTIMIZATIONS 6-29
6.7.1 Partial Memory Accesses 6-29
6.7.2 Increasing Bandwidth of Memory Fills and Video Fills. 6-30
6.7.2.1 Increasing Memory Bandwidth Using the MOVDQ Instruction 6-31
6.7.2.2 Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page . . 6-31
6.7.2.3 Increasing UC and WC Store Bandwidth by Using Aligned Stores 6-31
6.7.3 Reverse Memory Copy 6-31
6.8 CONVERTING FROM 64-BIT TO 128-BIT SIMD INTEGERS 6-34
6.8.1 SIMD Optimizations and Microarchitectures 6-35
6.8.1.1 Packed SSE2 Integer versus MMX Instructions 6-35
6.9 TUNING PARTIALLY VECTORIZABLE CODE 6-35
6.10 PARALLEL MODE AES ENCRYPTION AND DECRYPTION. 6-38
6.10.1 AES Counter Mode of Operation 6-39

6.10.2 AES Key Expansion Alternative 6-47
6.10.3 Enhancement in Haswell Microarchitecture 6-49
6.10.3.1 Advanced Encryption Standard (AES) and Multi-Buffer Cryptographic Throughput 6-49
6.11 LIGHT-WEIGHT DECOMPRESSION AND DATABASE PROCESSING 6-49
6.11.1 Reduced Dynamic Range Datasets 6-50
6.11.2 Compression and Decompression Using SIMD Instructions 6-50

CHAPTER 7
OPTIMIZING FOR SIMD FLOATING-POINT APPLICATIONS

7.1 GENERAL RULES FOR SIMD FLOATING-POINT CODE 7-1
7.2 PLANNING CONSIDERATIONS 7-1
7.3 USING SIMD FLOATING-POINT WITH X87 FLOATING-POINT 7-2
7.4 SCALAR FLOATING-POINT CODE 7-2
7.5 DATA ALIGNMENT 7-2
7.5.1 Data Arrangement 7-2
7.5.1.1 Vertical versus Horizontal Computation 7-3
7.5.1.2 Data Swizzling 7-5
7.5.1.3 Data Deswizzling 7-7
7.5.1.4 Horizontal ADD Using SSE 7-9
7.5.2 Use of CVTTPS2PI/CVTTSS2SI Instructions 7-11
7.5.3 Flush-to-Zero and Denormals-are-Zero Modes 7-11
7.6 SIMD OPTIMIZATIONS AND MICROARCHITECTURES 7-11
7.6.1 Dot Product and Horizontal SIMD Instructions 7-11
7.6.2 Vector Normalization 7-13
7.6.3 Using Horizontal SIMD Instruction Sets and Data Layout 7-15
7.6.3.1 SOA and Vector Matrix Multiplication 7-17

CHAPTER 8
INT8 DEEP LEARNING INFERENCE

8.1 INTRODUCING INT8 AS DATA TYPE FOR DEEP LEARNING INFERENCE 8-1
8.2 INTRODUCING INTEL® DL BOOST 8-1
8.2.1 Multiply and Add Unsigned and Signed Bytes (VPDPBUSD Instruction) 8-2
8.2.2 Multiply & Add Signed Word Integers (VPDPWSSD Instruction) 8-4
8.3 GENERAL OPTIMIZATIONS 8-4
8.3.1 Memory Layout 8-4
8.3.2 Quantization 8-5
8.3.2.1 Quantization of Weights 8-5
8.3.2.2 Quantization of Activations 8-5
8.3.2.3 Quantizing Negative Activations 8-6
8.3.3 Multicore Considerations 8-6
8.3.3.1 Large Batch (Throughput Workload) 8-6
8.3.3.2 Small Batch (Throughput at Latency Workload) 8-7
8.3.3.3 NUMA 8-7
8.4 CNNS 8-7
8.4.1 Convolutional Layers 8-7
8.4.1.1 Direct Convolution 8-7
8.4.1.2 Convolutional Layers with Low OFM Count 8-14
8.4.2 Post Convolution 8-16
8.4.2.1 Fused Quantization/Dequantization 8-16
8.4.2.2 ReLu 8-17
8.4.2.3 EltWise 8-17
8.4.2.4 Pooling 8-18
8.4.2.5 Pixel Shuffler 8-20
8.5 LSTM NETWORKS 8-22
8.5.1 Fused LSTM Embedding 8-22
8.5.2 Fused post GEMM 8-22

8.5.3 Dynamic Batch Size 8-25
8.5.4 NMT Example: Beam Search Decoder Get Top K 8-25

CHAPTER 9
OPTIMIZING CACHE USAGE

9.1 GENERAL PREFETCH CODING GUIDELINES 9-1
9.2 PREFETCH AND CACHEABILITY INSTRUCTIONS 9-2
9.3 PREFETCH 9-2
9.3.1 Software Data Prefetch 9-2
9.3.2 Prefetch Instructions 9-3
9.3.3 Prefetch and Load Instructions 9-5
9.4 CACHEABILITY CONTROL 9-5
9.4.1 The Non-temporal Store Instructions 9-5
9.4.1.1 Fencing 9-6
9.4.1.2 Streaming Non-temporal Stores 9-6
9.4.1.3 Memory Type and Non-temporal Stores 9-6
9.4.1.4 Write-Combining 9-6
9.4.2 Streaming Store Usage Models 9-7
9.4.2.1 Coherent Requests 9-7
9.4.2.2 Non-coherent requests 9-7
9.4.3 Streaming Store Instruction Descriptions 9-8
9.4.4 The Streaming Load Instruction 9-8
9.4.5 FENCE Instructions 9-8
9.4.5.1 SFENCE Instruction 9-8
9.4.5.2 LFENCE Instruction 9-9
9.4.5.3 MFENCE Instruction 9-9
9.4.6 CLFLUSH Instruction 9-9
9.4.7 CLFLUSHOPT Instruction 9-10
9.5 MEMORY OPTIMIZATION USING PREFETCH 9-12
9.5.1 Software-Controlled Prefetch 9-12
9.5.2 Hardware Prefetch 9-12
9.5.3 Example of Effective Latency Reduction with Hardware Prefetch 9-13
9.5.4 Example of Latency Hiding with S/W Prefetch Instruction 9-14
9.5.5 Software Prefetching Usage Checklist 9-15
9.5.6 Software Prefetch Scheduling Distance 9-16
9.5.7 Software Prefetch Concatenation 9-16
9.5.8 Minimize Number of Software Prefetches 9-17
9.5.9 Mix Software Prefetch with Computation Instructions 9-19
9.5.10 Software Prefetch and Cache Blocking Techniques 9-20
9.5.11 Hardware Prefetching and Cache Blocking Techniques 9-23
9.5.12 Single-Pass versus Multi-Pass Execution 9-24
9.6 MEMORY OPTIMIZATION USING NON-TEMPORAL STORES 9-25
9.6.1 Non-Temporal Stores and Software Write-Combining 9-25
9.6.2 Cache Management 9-26
9.6.2.1 Video Encoder 9-26
9.6.2.2 Video Decoder 9-26
9.6.2.3 Conclusions from Video Encoder and Decoder Implementation 9-27
9.6.2.4 Optimizing Memory Copy Routines 9-27
9.6.2.5 Using the 8-byte Streaming Stores and Software Prefetch 9-28
9.6.2.6 Using 16-byte Streaming Stores and Hardware Prefetch 9-29
9.6.2.7 Performance Comparisons of Memory Copy Routines 9-30
9.6.3 Deterministic Cache Parameters 9-30
9.6.3.1 Cache Sharing Using Deterministic Cache Parameters 9-32
9.6.3.2 Cache Sharing in Single-Core or Multicore 9-32
9.6.3.3 Determine Prefetch Stride 9-32

CHAPTER 10
SUB-NUMA CLUSTERING

10.1 SUB-NUMA CLUSTERING 10-1
10.2 COMPARISON WITH CLUSTER-ON-DIE 10-2
10.3 SNC USAGE 10-2
10.3.1 How to Check NUMA Configuration 10-2
10.3.2 MPI Optimizations for SNC 10-7
10.3.3 SNC Performance Comparison 10-8

CHAPTER 11
MULTICORE AND INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

11.1 PERFORMANCE AND USAGE MODELS 11-1
11.1.1 Multithreading 11-1
11.1.2 Multitasking Environment 11-3
11.2 PROGRAMMING MODELS AND MULTITHREADING 11-3
11.2.1 Parallel Programming Models 11-4
11.2.1.1 Domain Decomposition 11-4
11.2.2 Functional Decomposition 11-4
11.2.3 Specialized Programming Models 11-4
11.2.3.1 Producer-Consumer Threading Models 11-5
11.2.4 Tools for Creating Multithreaded Applications 11-8
11.2.4.1 Programming with OpenMP Directives 11-8
11.2.4.2 Automatic Parallelization of Code 11-8
11.2.4.3 Supporting Development Tools 11-8
11.3 OPTIMIZATION GUIDELINES 11-8
11.3.1 Key Practices of Thread Synchronization 11-9
11.3.2 Key Practices of System Bus Optimization 11-9
11.3.3 Key Practices of Memory Optimization 11-9
11.3.4 Key Practices of Execution Resource Optimization 11-10
11.3.5 Generality and Performance Impact 11-10
11.4 THREAD SYNCHRONIZATION 11-10
11.4.1 Choice of Synchronization Primitives 11-11
11.4.2 Synchronization for Short Periods 11-12
11.4.3 Optimization with Spin-Locks 11-13
11.4.4 Synchronization for Longer Periods 11-13
11.4.4.1 Avoid Coding Pitfalls in Thread Synchronization 11-14
11.4.5 Prevent Sharing of Modified Data and False-Sharing 11-15
11.4.6 Placement of Shared Synchronization Variable 11-16
11.5 SYSTEM BUS OPTIMIZATION 11-17
11.5.1 Conserve Bus Bandwidth 11-17
11.5.2 Understand the Bus and Cache Interactions 11-18
11.5.3 Avoid Excessive Software Prefetches 11-18
11.5.4 Improve Effective Latency of Cache Misses 11-19
11.5.5 Use Full Write Transactions to Achieve Higher Data Rate 11-19
11.6 MEMORY OPTIMIZATION 11-19
11.6.1 Cache Blocking Technique 11-20
11.6.2 Shared-Memory Optimization 11-20
11.6.2.1 Minimize Sharing of Data between Physical Processors 11-20
11.6.2.2 Batched Producer-Consumer Model 11-20
11.6.3 Eliminate 64-KByte Aliased Data Accesses 11-22
11.7 FRONT END OPTIMIZATION 11-22
11.7.1 Avoid Excessive Loop Unrolling 11-22
11.8 AFFINITIES AND MANAGING SHARED PLATFORM RESOURCES 11-22
11.8.1 Topology Enumeration of Shared Resources 11-24
11.8.2 Non-Uniform Memory Access (NUMA) 11-24
11.9 OPTIMIZATION OF OTHER SHARED RESOURCES 11-26
11.9.1 Expanded Opportunity for Intel® HT Optimization 11-26

CHAPTER 12
INTEL® OPTANE™ DC PERSISTENT MEMORY

12.1 MEMORY MODE AND APP-DIRECT MODE 12-1
12.1.1 Memory Mode. 12-1
12.1.2 App Direct Mode 12-1
12.1.3 Selecting a Mode. 12-2
12.2 DEVICE CHARACTERISTICS OF INTEL® OPTANE™ DC PERSISTENT MEMORY MODULE 12-4
12.2.1 Intel® Optane™ DC Persistent Memory Module Latency 12-4
12.2.2 Read vs. Write Bandwidth 12-4
12.2.3 Number of Threads for Optimal Bandwidth 12-6
12.3 PLATFORM IMPLICATIONS OF HANDLING A SECOND TYPE OF MEMORY. 12-8
12.3.1 Multi-Processor Cache Coherence. 12-8
12.3.2 Shared Queues in the Memory Hierarchy. 12-9
12.4 IMPLEMENTING PERSISTENCE FOR MEMORY 12-9
12.5 POWER CONSUMPTION. 12-10
12.5.1 Read-Write Equivalence 12-11
12.5.2 Spatial and Temporal Locality 12-12

CHAPTER 13
64-BIT MODE CODING GUIDELINES

13.1 INTRODUCTION 13-1
13.2 CODING RULES AFFECTING 64-BIT MODE 13-1
13.2.1 Use Legacy 32-Bit Instructions When Data Size Is 32 Bits 13-1
13.2.2 Use Extra Registers to Reduce Register Pressure 13-1
13.2.3 Effective Use of 64-Bit by 64-Bit Multiplication 13-2
13.2.4 Replace 128-bit Integer Division with 128-bit Multiplication 13-2
13.2.5 Sign Extension to Full 64-Bits 13-5
13.3 ALTERNATE CODING RULES FOR 64-BIT MODE 13-5
13.3.1 Use 64-Bit Registers Instead of Two 32-Bit Registers for 64-Bit Arithmetic Result 13-5
13.3.2 Using Software Prefetch 13-6

CHAPTER 14
INTEL® SSE4.2 AND SIMD PROGRAMMING FOR TEXT-PROCESSING/LEXING/PARSING

14.1 INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS 14-1
14.1.1 CRC32 14-4
14.2 USING INTEL® SSE4.2 STRING AND TEXT INSTRUCTIONS 14-5
14.2.1 Unaligned Memory Access and Buffer Size Management 14-6
14.2.2 Unaligned Memory Access and String Library 14-6
14.3 INTEL® SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES 14-6
14.3.1 Null Character Identification (Strlen equivalent) 14-7
14.3.2 White-Space-Like Character Identification 14-11
14.3.3 Substring Searches 14-13
14.3.4 String Token Extraction and Case Handling 14-19
14.3.5 Unicode Processing and PCMPxSTRy. 14-22
14.3.6 Replacement String Library Function Using Intel® SSE4.2 14-27
14.4 INTEL® SSE4.2-ENABLED NUMERICAL AND LEXICAL COMPUTATION 14-28
14.5 NUMERICAL DATA CONVERSION TO ASCII FORMAT. 14-34
14.5.1 Large Integer Numeric Computation. 14-48
14.5.1.1 MULX Instruction and Large Integer Numeric Computation 14-48

CHAPTER 15
OPTIMIZATIONS FOR INTEL® AVX, INTEL® AVX2, & INTEL® FMA

15.1 INTEL® AVX INTRINSICS CODING 15-2
15.1.1 Intel® AVX Assembly Coding 15-4
15.2 NON-DESTRUCTIVE SOURCE (NDS) 15-6
15.3 MIXING AVX CODE WITH SSE CODE 15-7
15.3.1 Mixing Intel® AVX and Intel SSE in Function Calls 15-11
15.4 128-BIT LANE OPERATION AND AVX 15-12
15.4.1 Programming With the Lane Concept 15-12
15.4.2 Strided Load Technique 15-13
15.4.3 The Register Overlap Technique 15-16
15.5 DATA GATHER AND SCATTER 15-18
15.5.1 Data Gather 15-18
15.5.2 Data Scatter 15-20
15.6 DATA ALIGNMENT FOR INTEL® AVX 15-22
15.6.1 Align Data to 32 Bytes 15-22
15.6.2 Consider 16-Byte Memory Access when Memory is Unaligned 15-23
15.6.3 Prefer Aligned Stores Over Aligned Loads 15-25
15.7 L1D CACHE LINE REPLACEMENTS 15-25
15.8 4K ALIASING 15-26
15.9 CONDITIONAL SIMD PACKED LOADS AND STORES 15-26
15.9.1 Conditional Loops 15-27
15.10 MIXING INTEGER AND FLOATING-POINT CODE 15-29
15.11 HANDLING PORT 5 PRESSURE 15-32
15.11.1 Replace Shuffles with Blends 15-32
15.11.2 Design Algorithm with Fewer Shuffles 15-34
15.11.3 Perform Basic Shuffles on Load Ports 15-37
15.12 DIVIDE AND SQUARE ROOT OPERATIONS 15-38
15.12.1 Single-Precision Divide 15-40
15.12.2 Single-Precision Reciprocal Square Root 15-42
15.12.3 Single-Precision Square Root 15-44
15.13 OPTIMIZATION OF ARRAY SUB SUM EXAMPLE 15-46
15.14 HALF-PRECISION FLOATING-POINT CONVERSIONS 15-48
15.14.1 Packed Single-Precision to Half-Precision Conversion 15-48
15.14.2 Packed Half-Precision to Single-Precision Conversion 15-50
15.14.3 Locality Consideration for using Half-Precision FP to Conserve Bandwidth 15-50
15.15 FUSED MULTIPLY-ADD (FMA) INSTRUCTIONS GUIDELINES 15-52
15.15.1 Optimizing Throughput with FMA and Floating-Point Add/MUL 15-52
15.15.2 Optimizing Throughput with Vector Shifts 15-54
15.16 AVX2 OPTIMIZATION GUIDELINES 15-55
15.16.1 Multi-Buffering and Intel® AVX2 15-60
15.16.2 Modular Multiplication and AVX2 15-60
15.16.3 Data Movement Considerations 15-60
15.16.3.1 SIMD Heuristics to implement Memcpy() 15-61
15.16.3.2 Memcpy() Implementation Using Enhanced REP MOVSB 15-62
15.16.3.3 Memset() Implementation Considerations 15-62
15.16.3.4 Hoisting Memcpy/Memset Ahead of Consuming Code 15-63
15.16.3.5 256-bit Fetch versus Two 128-bit Fetches 15-63
15.16.3.6 Mixing MULX and AVX2 Instructions 15-64
15.16.4 Considerations for Gather Instructions 15-70
15.16.4.1 Strided Loads 15-73
15.16.4.2 Adjacent Loads 15-75
15.16.5 Intel® AVX2 Conversion Remedy to MMX Instruction Throughput Limitation 15-76

CHAPTER 16
POWER OPTIMIZATION FOR MOBILE USAGES

16.1 OVERVIEW. 16-1
16.2 MOBILE USAGE SCENARIOS. 16-1
16.2.1 Intelligent Energy Efficient Software 16-2
16.3 ACPI C-STATES 16-3
16.3.1 Processor-Specific C4 and Deep C4 States 16-4
16.3.2 Processor-Specific Deep C-States and Intel® Turbo Boost Technology 16-5
16.3.3 Processor-Specific Deep C-States for Sandy Bridge Microarchitecture 16-5
16.3.4 Intel® Turbo Boost Technology 2.0 16-6
16.4 GUIDELINES FOR EXTENDING BATTERY LIFE 16-6
16.4.1 Adjust Performance to Meet Quality of Features 16-6
16.4.2 Reducing Amount of Work 16-7
16.4.3 Platform-Level Optimizations. 16-7
16.4.4 Handling Sleep State Transitions 16-8
16.4.5 Using Enhanced Intel SpeedStep® Technology. 16-8
16.4.6 Enabling Intel® Enhanced Deeper Sleep 16-9
16.4.7 Multicore Considerations 16-10
16.4.7.1 Enhanced Intel SpeedStep® Technology 16-10
16.4.7.2 Thread Migration Considerations 16-10
16.4.7.3 Multicore Considerations for C-States 16-11
16.5 TUNING SOFTWARE FOR INTELLIGENT POWER CONSUMPTION 16-12
16.5.1 Reduction of Active Cycles 16-12
16.5.1.1 Multithreading to Reduce Active Cycles 16-12
16.5.1.2 Vectorization 16-13
16.5.2 PAUSE and Sleep(0) Loop Optimization 16-14
16.5.3 Spin-Wait Loops. 16-15
16.5.4 Using Event Driven Service Instead of Polling in Code. 16-15
16.5.5 Reducing Interrupt Rate 16-15
16.5.6 Reducing Privileged Time 16-15
16.5.7 Setting Context Awareness in the Code 16-16
16.5.8 Saving Energy by Optimizing for Performance 16-17
16.6 PROCESSOR SPECIFIC POWER MANAGEMENT OPTIMIZATION FOR SYSTEM SOFTWARE 16-17
16.6.1 Power Management Recommendation of Processor-Specific Inactive State Configurations . . . 16-17
16.6.1.1 Balancing Power Management and Responsiveness of Inactive To Active State Transitions 16-19

CHAPTER 17
SOFTWARE OPTIMIZATION FOR INTEL® AVX-512 INSTRUCTIONS

17.1 BASIC INTEL® AVX-512 VS. INTEL® AVX2 CODING 17-2
17.1.1 Intrinsic Coding 17-3
17.1.2 Assembly Coding 17-5
17.2 MASKING. 17-7
17.2.1 Masking Example. 17-8
17.2.2 Masking Cost 17-12
17.2.3 Masking vs. Blending. 17-13
17.2.4 Nested Conditions / Mask Aggregation. 17-15
17.2.5 Memory Masking Microarchitecture Improvements. 17-16
17.2.6 Peeling and Remainder Masking 17-16
17.3 FORWARDING AND UNMASKED OPERATIONS 17-19
17.4 FORWARDING AND MEMORY MASKING 17-19
17.5 DATA COMPRESS 17-20
17.5.1 Data Compress Example 17-21
17.6 DATA EXPAND. 17-24
17.6.1 Data Expand Example 17-25
17.7 TERNARY LOGIC 17-27
17.7.1 Ternary Logic Example 1 17-27

17.7.2 Ternary Logic Example 2 17-29
17.8 NEW SHUFFLE INSTRUCTIONS 17-31
17.8.1 Two Source Permute Example 17-32
17.9 BROADCAST 17-34
17.9.1 Embedded Broadcast 17-34
17.9.2 Broadcast Executed on Load Ports 17-34
17.10 EMBEDDED ROUNDING 17-35
17.10.1 Static Rounding Mode 17-36
17.11 SCATTER INSTRUCTION 17-37
17.11.1 Data Scatter Example 17-38
17.12 STATIC ROUNDING MODES, SUPPRESS-ALL-EXCEPTIONS (SAE) 17-41
17.13 QWORD INSTRUCTION SUPPORT 17-41
17.13.1 QUADWORD Support in Arithmetic Instructions 17-41
17.13.2 QUADWORD Support in Convert Instructions 17-45
17.13.3 QUADWORD Support for Convert with Truncation Instructions 17-45
17.14 VECTOR LENGTH ORTHOGONALITY 17-46
17.15 INTEL® AVX-512 INSTRUCTIONS FOR TRANSCENDENTAL SUPPORT 17-46
17.15.1 VRCP14, VRSQRT14 - Software Sequences for 1/x, x/y, sqrt(x) 17-46
17.15.1.1 Application Examples 17-46
17.15.2 VGETMANT VGETEXP - Vector Get Mantissa and Vector Get Exponent 17-47
17.15.2.1 Application Examples 17-47
17.15.3 VRNDSCALE - Vector Round Scale 17-47
17.15.3.1 Application Examples 17-47
17.15.4 VREDUCE - Vector Reduce 17-48
17.15.4.1 Application Examples 17-48
17.15.5 VSCALEF - Vector Scale 17-48
17.15.5.1 Application Examples 17-48
17.15.6 VFPCLASS - Vector Floating Point Class 17-49
17.15.6.1 Application Examples 17-49
17.15.7 VPERM, VPERMI2, VPERMT2 - Small Table Lookup Implementation 17-49
17.15.7.1 Application Examples 17-49
17.16 CONFLICT DETECTION 17-50
17.16.1 Vectorization with Conflict Detection 17-50
17.16.2 Sparse Dot Product with VPCONFLICT 17-54
17.17 INTEL® AVX-512 VECTOR BYTE MANIPULATION INSTRUCTIONS (VBMI) 17-56
17.17.1 Permute Packet Bytes Elements Across Lanes (VPERMB) 17-56
17.17.2 Two-Source Byte Permute Across Lanes (VPERMI2B, VPERMT2B) 17-58
17.17.3 Select Packed Unaligned Bytes from Quadword Sources (VPMULTISHIFTQB) 17-61
17.18 FMA LATENCY 17-64
17.19 MIXING INTEL® AVX OR INTEL® AVX-512 EXTENSIONS WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) CODE 17-65
17.20 MIXING ZMM VECTOR CODE WITH XMM/YMM 17-66
17.21 SERVERS WITH A SINGLE FMA UNIT 17-67
17.22 GATHER/SCATTER TO SHUFFLE (G2S/STS) 17-71
17.22.1 Gather to Shuffle in Strided Loads 17-71
17.22.2 Scatter to Shuffle in Strided Stores 17-72
17.22.3 Gather to Shuffle in Adjacent Loads 17-73
17.23 DATA ALIGNMENT 17-75
17.23.1 Align Data to 64 Bytes 17-75
17.24 DYNAMIC MEMORY ALLOCATION AND MEMORY ALIGNMENT 17-76
17.25 DIVISION AND SQUARE ROOT OPERATIONS 17-77
17.25.1 Divide and Square Root Approximation Methods 17-78
17.25.2 Divide and Square Root Performance 17-78
17.25.3 Approximation Latencies 17-79
17.25.4 Code Snippets 17-82
17.26 CLDEMOTE 17-88
17.26.1 Producer-Consumer Communication in Software 17-88
17.27 TIPS ON COMPILER USAGE 17-89

CHAPTER 18
INTEL® ADVANCED VECTOR EXTENSIONS 512 - FP16 INSTRUCTION SET FOR INTEL® XEON® PROCESSORS

18.1 INTRODUCTION . . . 18-1
18.2 TERM DEFINITIONS . . . 18-1
18.3 OVERVIEW . . . 18-1
18.4 FP16 NUMERIC INSTRUCTIONS . . . 18-2
18.4.1 Data Type Support . . . 18-2
18.4.2 Overview of Intrinsics . . . 18-3
18.4.3 Fundamental Complex-Valued Support . . . 18-4
18.4.4 Using Intel® AVX-512 Bit Masks for Real-Valued Operations . . . 18-5
18.4.5 Using Intel® AVX-512 Bit Masks for Complex-Valued Operations . . . 18-6
18.5 NUMERICS . . . 18-9
18.5.1 Introduction to FP16 Number Format . . . 18-9
18.5.2 Observations on Representing Numbers in FP16 Format . . . 18-10
18.5.3 Numeric Accuracy Guarantees . . . 18-11
18.5.4 Handling Denormal Values . . . 18-12
18.5.5 Embedded Rounding . . . 18-13
18.5.6 Legacy FP16 Data Type Conversion . . . 18-13
18.5.7 FP16 Conversions to and from Other Data Types . . . 18-14
18.5.8 Approximation Instructions and Their Uses . . . 18-14
18.5.8.1 Approximate Reciprocal . . . 18-15
18.5.8.2 Approximate Division . . . 18-15
18.5.8.3 Approximate Reciprocal Square Root . . . 18-16
18.5.9 Approximate Square Root . . . 18-17
18.6 USING EXISTING INTEL® AVX-512 INSTRUCTIONS TO AUGMENT FP16 SUPPORT . . . 18-17
18.6.1 Using Existing Instructions to Extend Intel® AVX-512 FP16 Intrinsics . . . 18-17
18.6.2 Common Convenience Intrinsics . . . 18-18
18.6.3 Using Integer Comparisons for Fast Floating-Point Comparison . . . 18-18
18.7 MATH LIBRARY SUPPORT . . . 18-19

CHAPTER 19
CRYPTOGRAPHY & FINITE FIELD ARITHMETIC ENHANCEMENTS

19.1 VECTOR AES . . . 19-1
19.2 VPCLMULQDQ . . . 19-2
19.3 GALOIS FIELD NEW INSTRUCTIONS . . . 19-2
19.4 INTEGER FUSED MULTIPLY ACCUMULATE OPERATIONS (AVX512_IFMA - VPMADD52) . . . 19-4

CHAPTER 20
INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

20.1 DETECTING INTEL® AMX SUPPORT . . . 20-2
20.2 INTEL® AMX MICROARCHITECTURE OVERVIEW . . . 20-2
20.2.1 Intel® AMX Frequencies . . . 20-2
20.3 INTEL® AMX INSTRUCTIONS THROUGHPUT AND LATENCY . . . 20-3
20.4 DATA STRUCTURE ALIGNMENT . . . 20-3
20.5 GEMMS / CONVOLUTIONS . . . 20-3
20.5.1 Notation . . . 20-3
20.5.2 Tiles in the Intel® AMX Architecture . . . 20-4
20.5.3 B Matrix Layout . . . 20-6
20.5.4 Straightforward GEMM Implementation . . . 20-9
20.5.5 Optimizations . . . 20-10
20.5.5.1 Minimizing Tile Loads . . . 20-10
20.5.5.2 Software Pipelining of Tile Loads and Stores . . . 20-13
20.5.5.3 Optimized GEMM Implementation . . . 20-13
20.5.5.4 Direct Convolution with Intel® AMX . . . 20-15

20.5.5.5 Convolution - Matrix-like Multiplications and Summations Equivalence 20-17
20.5.5.6 Optimized Convolution Implementation 20-19
20.6 CACHE BLOCKING 20-21
20.6.1 Optimized Convolution Implementation with Cache Blocking 20-21
20.7 MINI-BATCHING IN LARGE BATCH INFERENCE 20-24
20.8 NON-TEMPORAL TILE LOADS 20-25
20.8.1 Priority Inversion Scenarios with Temporal Loads 20-26
20.9 USING LARGE TILES IN SMALL CONVOLUTIONS TO MAXIMIZE DATA REUSE 20-28
20.10 HANDLING INCONVENIENTLY-SIZED ACTIVATIONS 20-29
20.11 POST-CONVOLUTION OPTIMIZATIONS 20-29
20.11.1 Post-Convolution Fusion 20-29
20.11.2 Intel® AMX and Intel® AVX-512 Interleaving (SW Pipelining) 20-32
20.11.3 AVOIDING THE H/W OVERHEAD OF FREQUENT OPEN/CLOSE OPERATIONS in Port Five 20-35
20.11.4 Post-Convolution Multiple OFM Accumulation and Efficient Down-Conversion 20-35
20.12 INPUT AND OUTPUT BUFFERS REUSE (DOUBLE BUFFERING) 20-37
20.13 SOFTWARE PREFETCHES 20-39
20.13.1 Software Prefetch for Convolution and GEMM Layers 20-39
20.13.1.1 The Prefetch Strategy 20-39
20.13.1.2 Prefetch Distance 20-39
20.13.1.3 To Prefetch A or Prefetch B? 20-40
20.13.1.4 To Prefetch or Not to Prefetch C? 20-40
20.13.2 Software Prefetch for Embedding Layer 20-40
20.14 HARDWARE PREFETCHER TUNING 20-41
20.15 STORE TO LOAD FORWARDING 20-41
20.16 MATRIX TRANSPOSE 20-42
20.16.1 Flat-to-Flat Transpose of BF16 Data 20-42
20.16.2 VNNI-to-VNNI Transpose 20-46
20.16.3 Flat-to-VNNI Transpose 20-48
20.16.4 Flat-to-VNNI Re-layout 20-51
20.17 MULTI-THREADING CONSIDERATIONS 20-52
20.17.1 Thread Affinity 20-52
20.17.2 Intel® Hyper-Threading Technology (Intel® HT) 20-52
20.17.3 Work Partitioning Between Cores 20-53
20.17.3.1 Partitioning Over M 20-53
20.17.3.2 Partitioning Over N 20-53
20.17.3.3 Partitioning Over K 20-54
20.17.3.4 Memory Bandwidth Implications of Work Partitioning Over Multiple Dimensions 20-55
20.17.4 Recommendation System Example 20-56
20.18 SPARSITY OPTIMIZATIONS FOR INTEL® AMX 20-57
20.19 TILECONFIG/TILERELEASE, CORE C-STATE, AND COMPILER ABI 20-59
20.19.1 ABI 20-59
20.19.2 Intrinsics 20-60
20.19.3 User Interface 20-60
20.19.4 Intel® AMX Intrinsics Example 20-63
20.19.5 Compilation Option 20-65
20.20 INTEL® AMX STATE MANAGEMENT 20-65
20.20.1 Extended Feature Disable (XFD) 20-66
20.20.2 Alternate Signal Handler Stack in Linux Operating System 20-66
20.21 USING INTEL® AMX TO EMULATE HIGHER PRECISION GEMMS 20-66

CHAPTER 21
INTEL® QUICKASSIST TECHNOLOGY (INTEL® QAT)

21.1 SOFTWARE DESIGN GUIDELINES 21-1
21.1.1 Polling vs. Interrupts (If Supported) 21-1
21.1.1.1 Interrupt Mode 21-2
21.1.1.2 Polling Mode 21-2
21.1.1.3 Recommendations 21-3

21.1.2 Use of Data Plane (DP) API vs. Traditional API 21-3
21.1.2.1 Batch Submission of Requests Using the Data Plane API 21-3
21.1.3 Synchronous (sync) vs. Asynchronous (async) 21-4
21.1.4 Buffer Lists 21-4
21.1.5 Maximum Number of Concurrent Requests 21-4
21.1.6 Symmetric Crypto Partial Operations 21-5
21.1.7 Reusing Sessions in QAT Environment 21-5
21.1.8 Maximizing QAT Device Utilization 21-5
21.1.9 Best Known Method (BKM) for Avoiding Performance Bottlenecks 21-6
21.1.10 Avoid Data Copies By Using SVM and ATS 21-6
21.1.11 Avoid Page Faults When Using SVM 21-6

CHAPTER 22
USING PERFORMANCE MONITORING EVENTS

22.1 TOP-DOWN ANALYSIS METHOD 22-1
22.1.1 Top-Level 22-2
22.1.2 Frontend Bound 22-4
22.1.3 Backend Bound 22-4
22.1.4 Memory Bound 22-4
22.1.5 Core Bound 22-5
22.1.6 Bad Speculation 22-5
22.1.7 Retiring 22-6
22.1.8 Golden Cove Microarchitecture 22-6
22.1.9 Ice Lake Microarchitecture 22-6
22.1.10 Optane Persistent Memory 22-6
22.1.11 Skylake Microarchitecture 22-6
22.1.11.1 TMA Use Case 1 22-7
22.1.11.2 TMA Use Case 2 22-7
22.2 PERFORMANCE MONITORING AND MICROARCHITECTURE 22-8
22.3 INTEL® XEON® PROCESSOR 5500 SERIES 22-15
22.4 PERFORMANCE ANALYSIS TECHNIQUES FOR INTEL® XEON® PROCESSOR 5500 SERIES 22-16
22.4.1 Cycle Accounting and Uop Flow Analysis 22-17
22.4.1.1 Cycle Drill Down and Branch Mispredictions 22-18
22.4.1.2 Basic Block Drill Down 22-21
22.4.2 Stall Cycle Decomposition and Core Memory Accesses 22-22
22.4.2.1 Measuring Costs of Microarchitectural Conditions 22-22
22.4.3 Core PMU Precise Events 22-23
22.4.3.1 Precise Memory Access Events 22-24
22.4.3.2 Load Latency Event 22-25
22.4.3.3 Precise Execution Events 22-27
22.4.3.4 Last Branch Record (LBR) 22-28
22.4.3.5 Measuring Per-Core Bandwidth 22-33
22.4.3.6 Miscellaneous L1 and L2 Events for Cache Misses 22-34
22.4.3.7 TLB Misses 22-34
22.4.3.8 L1 Data Cache 22-34
22.4.4 Front End Monitoring Events 22-35
22.4.4.1 Branch Mispredictions 22-35
22.4.4.2 Front End Code Generation Metrics 22-35
22.4.5 Uncore Performance Monitoring Events 22-36
22.4.5.1 Global Queue Occupancy 22-36
22.4.5.2 Global Queue Port Events 22-38
22.4.5.3 Global Queue Snoop Events 22-38
22.4.5.4 L3 Events 22-39
22.4.6 Intel QuickPath Interconnect Home Logic (QHL) 22-39
22.4.7 Measuring Bandwidth From the Uncore 22-45
22.5 PERFORMANCE TUNING TECHNIQUES FOR SANDY BRIDGE MICROARCHITECTURE 22-45
22.5.1 Correlating Performance Bottleneck to Source Location 22-45

22.5.2 Hierarchical Top-Down Performance Characterization Methodology and Locating Performance Bottlenecks 22-46
22.5.2.1 Back End Bound Characterization 22-47
22.5.2.2 Core Bound Characterization 22-47
22.5.2.3 Memory Bound Characterization 22-48
22.5.3 Back End Stalls 22-49
22.5.4 Memory Sub-System Stalls 22-50
22.5.4.1 Accounting for Load Latency 22-50
22.5.4.2 Cache-line Replacement Analysis 22-52
22.5.4.3 Lock Contention Analysis 22-52
22.5.4.4 Other Memory Access Issues 22-53
22.5.5 Execution Stalls 22-56
22.5.5.1 Longer Instruction Latencies 22-56
22.5.5.2 Assists 22-57
22.5.6 Bad Speculation 22-57
22.5.6.1 Branch Mispredicts 22-57
22.5.7 Frontend Stalls 22-58
22.5.7.1 Understanding the Micro-op Delivery Rate 22-58
22.5.7.2 Understanding the Sources of the Micro-op Queue 22-59
22.5.7.3 The Decoded ICache 22-61
22.5.7.4 Issues in the Legacy Decode Pipeline 22-62
22.5.7.5 Instruction Cache 22-62
22.6 USING PERFORMANCE EVENTS OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS 22-62
22.6.1 Understanding the Results in a Performance Counter 22-63
22.6.2 Ratio Interpretation 22-63
22.6.3 Notes on Selected Events 22-64
22.7 DRILL-DOWN TECHNIQUES FOR PERFORMANCE ANALYSIS 22-64
22.7.1 Cycle Composition at Issue Port 22-66
22.7.2 Cycle Composition of OOO Execution 22-66
22.7.3 Drill-Down on Performance Stalls 22-67
22.8 EVENT RATIOS FOR INTEL CORE MICROARCHITECTURE 22-68
22.8.1 Clocks Per Instructions Retired Ratio (CPI) 22-68
22.8.1.1 Front End Ratios 22-68
22.8.1.2 Branching and Front End 22-69
22.8.1.3 Stack Pointer Tracker 22-69
22.8.1.4 Macro-fusion 22-69
22.8.1.5 Length Changing Prefix (LCP) Stalls 22-69
22.8.1.6 Self Modifying Code Detection 22-70
22.8.2 Branch Prediction Ratios 22-70
22.8.2.1 Branch Mispredictions 22-70
22.8.2.2 Virtual Tables and Indirect Calls 22-70
22.8.2.3 Mispredicted Returns 22-70
22.8.3 Execution Ratios 22-71
22.8.3.1 Resource Stalls 22-71
22.8.3.2 ROB Read Port Stalls 22-71
22.8.3.3 Partial Register Stalls 22-71
22.8.3.4 Partial Flag Stalls 22-71
22.8.3.5 Bypass Between Execution Domains 22-71
22.8.3.6 Floating-Point Performance Ratios 22-71
22.8.4 Memory Sub-System - Access Conflicts Ratios 22-72
22.8.4.1 Loads Blocked by the L1 Data Cache 22-72
22.8.4.2 4K Aliasing and Store Forwarding Block Detection 22-72
22.8.4.3 Load Block by Preceding Stores 22-72
22.8.4.4 Memory Disambiguation 22-73
22.8.4.5 Load Operation Address Translation 22-73
22.8.5 Memory Sub-System - Cache Misses Ratios 22-73
22.8.5.1 Locating Cache Misses in the Code 22-73
22.8.5.2 L1 Data Cache Misses 22-73

22.8.5.3 L2 Cache Misses 22-73
22.8.6 Memory Sub-system - Prefetching 22-74
22.8.6.1 L1 Data Prefetching 22-74
22.8.6.2 L2 Hardware Prefetching 22-74
22.8.6.3 Software Prefetching 22-74
22.8.7 Memory Sub-system - TLB Miss Ratios 22-74
22.8.8 Memory Sub-system - Core Interaction 22-75
22.8.8.1 Modified Data Sharing 22-75
22.8.8.2 Fast Synchronization Penalty 22-75
22.8.8.3 Simultaneous Extensive Stores and Load Misses 22-75
22.8.9 Memory Sub-system - Bus Characterization 22-75
22.8.9.1 Bus Utilization 22-75
22.8.9.2 Modified Cache Lines Eviction 22-76

APPENDIX A
APPLICATION PERFORMANCE TOOLS

A.1 COMPILERS A-2
A.1.1 Recommended Optimization Settings for Intel® 64 and IA-32 Processors A-2
A.1.2 Vectorization and Loop Optimization A-3
A.1.2.1 Multithreading with OpenMP* A-3
A.1.2.2 Automatic Multithreading A-3
A.1.3 Inline Expansion of Library Functions (/Oi, /Oi-) A-3
A.1.4 Interprocedural and Profile-Guided Optimizations A-3
A.1.4.1 Interprocedural Optimization (IPO) A-4
A.1.4.2 Profile-Guided Optimization (PGO) A-4
A.1.5 Intel® Cilk™ Plus A-4
A.2 PERFORMANCE LIBRARIES A-4
A.2.1 Intel® Integrated Performance Primitives (Intel® IPP) A-5
A.2.2 Intel® Math Kernel Library (Intel® MKL) A-5
A.2.3 Intel® Threading Building Blocks (Intel® TBB) A-5
A.2.4 Benefits Summary A-5
A.3 PERFORMANCE PROFILERS A-6
A.3.1 Intel® VTune™ Amplifier XE A-6
A.3.1.1 Hardware Event-Based Sampling Analysis A-6
A.3.1.2 Algorithm Analysis A-6
A.3.1.3 Platform Analysis A-7
A.4 THREAD AND MEMORY CHECKERS A-7
A.4.1 Intel® Inspector A-7
A.5 VECTORIZATION ASSISTANT A-7
A.5.1 Intel® Advisor A-7
A.6 CLUSTER TOOLS A-7
A.6.1 Intel® Trace Analyzer and Collector A-7
A.6.1.1 MPI Performance Snapshot A-8
A.6.2 Intel® MPI Library A-8
A.6.3 Intel® MPI Benchmarks A-8
A.7 INTEL® COMMUNITIES A-8

APPENDIX B
RUNTIME PERFORMANCE OPTIMIZATION BLUEPRINT: INTEL® ARCHITECTURE OPTIMIZATION WITH LARGE CODE PAGES

B.1 OVERVIEW B-1
B.1.1 TLBs and Stalls B-1
B.1.2 Large Pages B-2
B.2 DIAGNOSING THE PROBLEM B-2
B.2.1 ITLB Misses B-2
B.2.2 Measuring the ITLB Miss Stall B-5

B.2.3 Source of ITLB Misses B-6
B.3 SOLUTION B-7
B.3.1 Linux* and Large Pages B-7
B.3.2 Large Pages for .text B-7
B.3.3 Reference Code B-8
B.3.4 Large Pages for the Heap B-9
B.4 SOLUTION INTEGRATION B-10
B.4.1 V8 Integration with the Reference Implementation B-10
B.4.2 JAVA JVM Integration with the Reference Implementation B-10
B.5 LIMITATIONS B-11
B.6 CASE STUDY B-11
B.6.1 Ghost.js Workload B-12
B.6.2 Web Tooling Workload B-12
B.6.2.1 Node Version B-12
B.6.2.2 Web Tooling B-13
B.6.2.3 Comparing Clear Linux* OS and Ubuntu* B-13
B.6.3 MediaWiki Workload B-13
B.6.4 Visualization of Benefits B-14
B.6.4.1 Precise Events B-14
B.6.4.2 Visualizing Precise ITLB Miss B-14
B.7 SUMMARY B-17
B.8 TEST CONFIGURATION DETAILS B-17
B.9 ADDITIONAL REFERENCES B-19

TABLES

PAGE

Table 1-1. Intel Processors Organized by Microarchitecture 1-2
Table 1-2. Term Definitions 1-6
Table 1-3. Additional References in and Beyond this Document 1-7
Table 2-1. MOV+OP Macro-Fusion Example 2-3
Table 2-2. LD+OP Macro-Fusion Example 2-3
Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture 2-12
Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions 2-13
Table 2-5. Bypass Delay Between Producer and Consumer Micro-Ops 2-14
Table 2-6. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture 2-19
Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions 2-19
Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops 2-20
Table 2-9. Cache Parameters of the Ice Lake Client Microarchitecture 2-21
Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture 2-21
Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture . . 2-27
Table 2-12. Maximum Intel® Turbo Boost Technology Core Frequency Levels 2-30
Table 2-13. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture 2-34
Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions 2-34
Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops 2-35
Table 2-16. Cache Parameters of the Skylake Client Microarchitecture 2-36
Table 2-17. TLB Parameters of the Skylake Client Microarchitecture 2-37
Table 3-1. Macro-Fusible Instructions in the Sandy Bridge Microarchitecture 3-13
Table 3-2. Macro-Fusible Instructions in the Haswell Microarchitecture 3-13
Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction 3-27
Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture 3-48
Table 3-5. Relative Performance of Memcpy() Using Enhanced REP MOVSB and STOSB Vs. 128-bit AVX3-65
Table 3-6. Effect of Address Misalignment on Memcpy() Performance 3-65
Table 3-7. Intel Processor CPU RP Device IDs for Processors Optimizing PCIe Performance 3-73
Table 3-8. Samples: 365K of Events 'anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}', Event Count (a--r0x): 679008523-74
Table 3-9. Shared Data Cache Line Table 3-79
Table 3-10. Shared Cache Line Distribution Pareto 3-79
Table 3-11. False Sharing Improvements 3-80
Table 3-12. Instruction Sequence Mixing VEX on the Sapphire Rapids and Ice Lake Server Microarchitectures3-81
Table 3-13. Fixed Instruction Sequence with Improved Performance on Sapphire Rapids Microarchitecture3-82
Table 3-14. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction 3-83
Table 4-1. Paging Cache Parameters of the Crestmont Microarchitecture 4-7
Table 4-2. Dispatch Port and Execution Stacks of the Tremont Microarchitecture 4-14
Table 4-3. Cache Parameters of the Tremont Microarchitecture 4-15
Table 6-1. PSHUF Instruction Byte Encoding of Individual Words 6-13
Table 7-1. SoA Form of Representing Vertices Data 7-4
Table 9-1. Implementation Details of Prefetch Hint Instructions for Intel® Core™ Duo, Intel® Core™ 2, and Intel® Atom® Processors9-4
Table 9-2. Details of Prefetch Hint Instructions for Processors Based on Skylake Microarchitectures9-4
Table 9-3. Implementation Details of Prefetch Hint Instructions for Intel® Xeon® Scalable Family of Processors9-4
Table 9-4. Software Prefetching Considerations into Strip-mining Code 9-23
Table 9-5. Deterministic Cache Parameters Leaf 9-31
Table 11-1. Properties of Synchronization Objects 11-11
Table 11-2. Design-Time Resource Management Choices 11-23
Table 11-3. Microarchitectural Resources Comparisons of Intel® HT Implementations 11-26

TABLES

PAGE

Table 12-1. Latencies for Accessing Intel® Optane™ DC Persistent Memory Modules 12-4
Table 12-2. Bandwidths per DIMM for Intel® Optane™ DC Persistent Memory Modules and DRAM 12-5
Table 14-1. Intel® SSE4.2 String/Text Instructions Compare Operation on N-Elements 14-3
Table 14-2. Intel® SSE4.2 String/Text Instructions Unary Transformation on IntRes1. 14-3
Table 14-3. Intel® SSE4.2 String/Text Instructions Output Selection Imm[6] 14-3
Table 14-4. SSE4.2 String/Text Instructions Element-Pair Comparison Definition. 14-4
Table 14-5. SSE4.2 String/Text Instructions Eflags Behavior 14-4
Table 15-1. Features Between 256-bit Intel® AVX, 128-bit Intel® AVX, and Legacy Intel® SSE Extensions. 15-2
Table 15-2. State Transitions of Mixing AVX and SSE Code 15-10
Table 15-3. Approximate Magnitude of Intel® AVX—Intel® SSE Transition Penalties in Different Microarchitectures. 15-10
Table 15-4. Effect of VZEROUPPER with Inter-Function Calls Between AVX and SSE Code. 15-12
Table 15-5. Comparison of Numeric Alternatives of Selected Linear Algebra in Skylake Microarchitecture 15-39
Table 15-6. Single-Precision Divide and Square Root Alternatives 15-40
Table 15-7. Comparison of AOS to SOA with Strided Access Pattern 15-74
Table 15-8. Comparison of Indexed AOS to SOA Transformation 15-76
Table 16-1. ACPI C-State-Type Mappings to Processor Specific C-State of the Sandy Bridge Microarchitecture 16-5
Table 16-2. C-State Total Processor Exit Latency for Client Systems with Slow VR 16-18
Table 16-3. C-State Total Processor Exit Latency for Client Systems (Core+ Package Exit Latency) with Fast VR 16-18
Table 16-4. C-State Core-Only Exit Latency for Client Systems with Slow VR 16-19
Table 16-5. POWER_CTL MSR in Processors Based on Sandy Bridge Microarchitecture. 16-19
Table 17-1. Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations 17-1
Table 17-2. Cache Comparison Between Skylake Server Microarchitecture and Broadwell Microarchitecture 17-16
Table 17-3. Static Rounding Mode Functions 17-36
Table 17-4. Software Sequence and Hardware Scatter 17-39
Table 17-5. Vector Quadword Extensions 17-45
Table 17-6. Scalar Quadword Extensions. 17-45
Table 17-7. Vector Quadword Extensions 17-45
Table 17-8. Scalar Quadword Extensions. 17-45
Table 17-9. FMA Unit Latency. 17-64
Table 17-10. Data Alignment Effects on SAXPY Performance vs. Speedup Value 17-76
Table 17-11. Skylake Microarchitecture Recommendations for DIV/SQRT-Based Operations (Single-Precision) 17-78
Table 17-12. Skylake Microarchitecture Recommendations for DIV/SQRT-Based Operations (Double-Precision) 17-78
Table 17-13. 256-bit Intel AVX2 Divide and Square Root Instruction Performance. 17-79
Table 17-14. 512-bit Intel AVX-512 Divide and Square Root Instruction Performance 17-79
Table 17-15. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Single Precision 17-80
Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision 17-80
Table 18-1. Supported FP16 Data Types 18-2
Table 18-2. Example Intrinsic Names 18-3
Table 18-3. Conjugation Instructions 18-5
Table 18-4. Useful or Interesting FP16 Numbers 18-11
Table 18-5. Conjugation Instructions 18-13
Table 18-6. Conjugation Instructions 18-18
Table 20-1. Intel® AMX-Related Links. 20-1
Table 20-2. Intel® AMX Instruction Throughput and Latency 20-3
Table 20-3. Five Loops in Example 20-20 20-25

TABLES

PAGE

Table 20-4. Accessed Data Sizes: Scenario One 20-26
Table 20-5. Accessed Data Sizes: Scenario Two 20-26
Table 20-6. Accessed Data Sizes Extended to Blocked Code 20-27
Table 20-7. Loading 32 Quarter-Cache Lines into 8 ZMM Registers 20-45
Table 20-8. Loading Eight Quarter-Cache Lines into Two ZMM Registers 20-48
Table 20-9. BF16 Flat-to-VNNI Transpose 20-50
Table 20-10. A Simple Partition of Work Between Three Threads 20-55
Table 20-11. An Optimized Partition of Work Between Three Threads 20-56
Table 22-1. Performance Monitoring Taxonomy 22-9
Table 22-2. Cycle Accounting and Micro-ops Flow Recipe 22-17
Table 22-3. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow 22-18
Table 22-4. Cycle Accounting of Wasted Work Due to Misprediction 22-19
Table 22-5. Cycle Accounting of Instruction Starvation 22-20
Table 22-6. CMask/Inv/Edge/Thread Granularity of Events for Micro-op Flow 22-21
Table 22-7. Approximate Latency of L2 Misses of Intel Xeon Processor 5500 22-23
Table 22-8. Load Latency Event Programming 22-25
Table 22-9. Data Source Encoding for Load Latency PEBS Record 22-26
Table 22-10. Core PMU Events to Drill Down L2 Misses 22-31
Table 22-11. Core PMU Events for Super Queue Operation 22-31
Table 22-12. Core PMU Event to Drill Down OFFCore Responses 22-31
Table 22-13. OFFCORE_RSP_0 MSR Programming 22-32
Table 22-14. Common Request and Response Types for OFFCORE_RSP_0 MSR 22-32
Table 22-15. Uncore PMU Events for Occupancy Cycles 22-38
Table 22-16. Common QHL Opcode Matching Facility Programming 22-40
Table A-1. Recommended Processor Optimization Options A-2
Table B-1. Core TLB Structure Size and Organization Across Multiple Intel Product Generations B-2
Table B-2. Calculating ITLB Miss Stall for Ghost.js B-3
Table B-3. ITLB MPKI and Executable Sizes Across Various Workloads B-5
Table B-4. Key Metrics for Ghost.js With and Without Large Pages B-12
Table B-5. Key Metrics for Web Tooling across Clear Linux and Ubuntu 18.04 . . . . . B-13
Table B-6. Key Metrics for MediaWiki Workload on HHVM B-14
Table B-7. Precise Front-end Events for ITLB Misses B-14
Table B-8. System Details B-17
Table B-9. Processor Information B-18
Table B-10. Kernel Vulnerability Status B-18

EXAMPLES

Example 2-1. Class 0 Pseudo-code Snippet 2-7
Example 2-2. Class 1 Pseudo-code Snippet 2-8
Example 2-3. Class 2 Pseudo-code Snippet 2-9
Example 2-4. Class 3 Pseudo-code Snippet 2-9
Example 2-5. Breaking False Dependency through Zero Idiom 2-16
Example 2-6. Considering Stores 2-22
Example 2-7. Rearranging Code to Achieve Store Pairing 2-23
Example 2-8. FSFP Optimization 2-24
Example 2-9. Dynamic Pause Loop Example 2-38
Example 2-10. Contended Locks with Increasing Back-off Example 2-39
Example 3-1. Assembly Code With an Unpredictable Branch 3-5
Example 3-2. Code Optimization to Eliminate Branches 3-5
Example 3-3. Eliminating Branch with CMOV Instruction 3-6
Example 3-4. Static Branch Prediction Algorithm 3-6
Example 3-5. Static Taken Prediction 3-7
Example 3-6. Static Not-Taken Prediction 3-7
Example 3-7. Indirect Branch with Two Favored Targets 3-9
Example 3-8. A Peeling Technique to Reduce Indirect Branch Misprediction 3-10
Example 3-9. Loop Unrolling 3-11
Example 3-10. Macrofusion, Unsigned Iteration Count 3-14
Example 3-12. Macrofusion, Signed Variable 3-15
Example 3-11. Macrofusion, If Statement 3-15
Example 3-13. Macrofusion, Signed Comparison 3-16
Example 3-14. Additional Macrofusion Benefit in the Sandy Bridge Microarchitecture 3-16
Example 3-15. Avoiding False LCP Delays with 0xF7 Group Instructions 3-18
Example 3-16. Independent Two-Operand LEA Example 3-22
Example 3-17. Alternative to Three-Operand LEA 3-22
Example 3-18. Examples of 512-bit Additions 3-23
Example 3-19. Clearing Register to Break Dependency While Negating Array Elements 3-26
Example 3-20. Spill Scheduling Code 3-28
Example 3-21. Zero-Latency MOV Instructions 3-29
Example 3-22. Byte-Granular Data Computation Technique 3-30
Example 3-23. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions 3-30
Example 3-24. Avoiding Partial Register Stalls in SIMD Code 3-33
Example 3-25. Avoiding Partial Flag Register Stalls 3-34
Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture 3-34
Example 3-27. Reference Code Template for Partially Vectorizable Program 3-36
Example 3-28. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty 3-37
Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing 3-38
Example 3-30. Stack Optimization Technique to Simplify Parameter Passing 3-38
Example 3-31. Base Line Code Sequence to Estimate Loop Overhead 3-39
Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture 3-41
Example 3-33. Index versus Pointers in Pointer-Chasing Code 3-42
Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy 3-43
Example 3-35. Using XMM Register in Lieu of Memory for Register Spills 3-44
Example 3-36. Loads Blocked by Stores of Unknown Address 3-45
Example 3-37. Situations Showing Small Loads After Large Store 3-47
Example 3-38. Non-forwarding Example of Large Load After Small Store 3-47
Example 3-39. A Non-forwarding Situation in Compiler Generated Code 3-47
Example 3-40. Two Ways to Avoid Non-forwarding Situation in Example 3-39 3-48
Example 3-41. Large and Small Load Stalls 3-48
Example 3-42. Loop-Carried Dependence Chain 3-50
Example 3-43. Rearranging a Data Structure 3-51
Example 3-44. Decomposing an Array 3-51

EXAMPLES

Example 3-45. Examples of Dynamical Stack Alignment 3-53
Example 3-46. Instruction Pointer Query Techniques 3-54
Example 3-47. Using Non-Temporal Stores and 64-byte Bus Write Transactions 3-56
Example 3-48. On-temporal Stores and Partial Bus Write Transactions 3-57
Example 3-49. Using DCU Hardware Prefetch 3-58
Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines 3-59
Example 3-51. Technique for Using L1 Hardware Prefetch 3-60
Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination 3-62
Example 3-53. Algorithm to Avoid Changing Rounding Mode 3-70
Example 3-54. Locking Algorithm for the Sapphire Rapids Microarchitecture 3-76
Example 3-55. Perf Annotation for runtime.getempty 3-78
Example 3-56. Padding Insertion in Go Runtime 3-79
Example 3-57. Identifying >2GB Branches 3-83
Example 3-58. Identification of WAITPKG with CPUID 3-86
Example 3-59. Code Snippet in an Asynchronous Example 3-87
Example 5-1. Identification of MMX Technology with CPUID 5-2
Example 5-2. Identification of Intel® SSE with CPUID 5-2
Example 5-3. Identification of Intel® SSE2 with cpuid 5-3
Example 5-4. Identification of Intel® SSE3 with CPUID 5-3
Example 5-5. Identification of SSSE3 with cpuid 5-4
Example 5-6. Identification of Intel® SSE4.1 with CPUID 5-4
Example 5-7. Identification of SSE4.2 with cpuid 5-4
Example 5-8. Detection of AESNI Instructions 5-5
Example 5-9. Detection of PCLMULQDQ Instruction 5-5
Example 5-10. Detection of Intel® AVX Instruction 5-6
Example 5-11. Detection of VEX-Encoded AESNI Instructions 5-7
Example 5-12. Detection of VEX-Encoded AESNI Instructions 5-7
Example 5-13. Simple Four-Iteration Loop 5-15
Example 5-14. Intel® Streaming SIMD Extensions (Intel® SSE) Using Inlined Assembly Encoding 5-15
Example 5-15. Simple Four-Iteration Loop Coded with Intrinsics 5-16
Example 5-16. C++ Code Using the Vector Classes 5-17
Example 5-17. Automatic Vectorization for a Simple Loop 5-17
Example 5-18. C Algorithm for 64-bit Data Alignment 5-20
Example 5-19. AoS Data Structure 5-22
Example 5-20. SoA Data Structure 5-22
Example 5-21. AoS and SoA Code Samples 5-23
Example 5-22. Hybrid SoA Data Structure 5-24
Example 5-23. Pseudo-Code Before Strip Mining 5-25
Example 5-24. Strip Mined Code 5-26
Example 5-25. Loop Blocking 5-26
Example 5-26. Emulation of Conditional Moves 5-29
Example 6-1. Resetting Register Between __m64 and FP Data Types Code 6-3
Example 6-2. FIR Processing Example in C language Code 6-4
Example 6-3. SSE2 and SSSE3 Implementation of FIR Processing Code 6-4
Example 6-4. Zero Extend 16-bit Values into 32 Bits Using Unsigned Unpack Instructions Code 6-5
Example 6-5. Signed Unpack Code 6-6
Example 6-6. Interleaved Pack with Saturation Code 6-7
Example 6-7. Interleaved Pack without Saturation Code 6-8
Example 6-8. Unpacking Two Packed-word Sources in Non-Interleaved Way Code 6-9
Example 6-9. PEXTRW Instruction Code 6-10
Example 6-10. PINSRW Instruction Code 6-11
Example 6-11. Repeated PINSRW Instruction Code 6-11
Example 6-12. Non-Unit Stride Load/Store Using SSE4.1 Instructions 6-12
Example 6-13. Scatter and Gather Operations Using SSE4.1 Instructions 6-12

EXAMPLES

Example 6-14. PMOVMSKB Instruction Code 6-13
Example 6-15. Broadcast a Word Across XMM, Using 2 SSE2 Instructions 6-14
Example 6-16. Swap/Reverse Words in an XMM, Using 3 SSE2 Instructions 6-14
Example 6-17. Generating Constants 6-15
Example 6-18. Absolute Difference of Two Unsigned Numbers 6-16
Example 6-19. Absolute Difference of Signed Numbers 6-17
Example 6-20. Computing Absolute Value 6-17
Example 6-21. Basic C Implementation of RGBA to BGRA Conversion 6-18
Example 6-22. Color Pixel Format Conversion Using SSE2 6-18
Example 6-23. Color Pixel Format Conversion Using SSSE3 6-19
Example 6-24. Big-Endian to Little-Endian Conversion 6-20
Example 6-25. Clipping to a Signed Range of Words [High, Low] 6-21
Example 6-26. Clipping to an Arbitrary Signed Range [High, Low] 6-21
Example 6-27. Simplified Clipping to an Arbitrary Signed Range 6-21
Example 6-28. Clipping to an Arbitrary Unsigned Range [High, Low] 6-22
Example 6-29. Complex Multiply by a Constant 6-24
Example 6-30. Using PTEST to Separate Vectorizable and Non-Vectorizable Loop Iterations 6-25
Example 6-31. Using Variable BLEND to Vectorize Heterogeneous Loops 6-26
Example 6-32. Baseline C Code for Mandelbrot Set Map Evaluation 6-26
Example 6-33. Vectorized Mandelbrot Set Map Evaluation Using SSE4.1 Intrinsics 6-27
Example 6-34. A Large Load after a Series of Small Stores (Penalty) 6-29
Example 6-35. Accessing Data without Delay 6-29
Example 6-36. A Series of Small Loads after a Large Store 6-30
Example 6-37. Eliminating Delay for a Series of Small Loads after a Large Store 6-30
Example 6-38. Un-optimized Reverse Memory Copy in C 6-32
Example 6-39. Using PSHUFB to Reverse Byte Ordering 16 Bytes at a Time 6-33
Example 6-40. Table Look-up Operations in C Code 6-35
Example 6-41. Shift Techniques on Non-Vectorizable Table Look-up 6-36
Example 6-42. PEXTRD Techniques on Non-Vectorizable Table Look-up 6-37
Example 6-43. Pseudo-Code Flow of AES Counter Mode Operation 6-39
Example 6-44. AES128-CTR Implementation with Eight Block in Parallel 6-40
Example 6-45. AES128 Key Expansion 6-48
Example 6-46. Compress 32-bit Integers into 5-bit Buckets 6-51
Example 6-47. Decompression of a Stream of 5-bit Integers into 32-bit Elements 6-52
Example 7-1. Pseudocode for Horizontal (xyz, AoS) Computation 7-5
Example 7-2. Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation 7-5
Example 7-3. Swizzling Data Using SHUFPS, MOVLHPS, MOVHLPS 7-5
Example 7-4. Swizzling Data Using UNPCKxxx Instructions 7-6
Example 7-5. Deswizzling Single-Precision SIMD Data 7-7
Example 7-6. Deswizzling Data Using SIMD Integer Instructions 7-8
Example 7-7. Horizontal Add Using MOVHLPS/MOVLHPS 7-10
Example 7-8. Horizontal Add Using Intrinsics with MOVHLPS/MOVLHPS 7-10
Example 7-9. Dot Product of Vector Length 4 Using SSE/SSE2 7-12
Example 7-10. Dot Product of Vector Length 4 Using SSE3 7-12
Example 7-11. Dot Product of Vector Length 4 Using SSE4.1 7-12
Example 7-12. Unrolled Implementation of Four Dot Products 7-13
Example 7-13. Normalization of an Array of Vectors 7-13
Example 7-14. Normalize (x, y, z) Components of an Array of Vectors Using SSE2 7-14
Example 7-15. Normalize (x, y, z) Components of an Array of Vectors Using SSE4.1 7-15
Example 7-16. Data Organization in Memory for AOS Vector-Matrix Multiplication 7-16
Example 7-17. AOS Vector-Matrix Multiplication with HADDPS 7-16
Example 7-18. AOS Vector-Matrix Multiplication with DPPS 7-17
Example 7-19. Data Organization in Memory for SOA Vector-Matrix Multiplication 7-18
Example 7-20. Vector-Matrix Multiplication with Native SOA Data Layout 7-19

EXAMPLES

Example 8-1. VPDPBUSD Implementation 8-3
Example 8-2. Quantization of Activations (Scalar) 8-5
Example 8-3. Quantization of Activations (Vectorized) 8-5
Example 8-4. Direct Convolution 8-12
Example 8-5. Convolution for Layers with Low OFM Count 8-15
Example 8-6. Basic PostConv 8-16
Example 8-7. Uint8 Residual Input 8-18
Example 8-8. 8x8 Average Pooling with Stride 1 of 8x8 Layers 8-18
Example 8-9. Unfused Vectorized Pooling 8-19
Example 8-10. Caffe Scalar Code for Pixel Shuffler 8-20
Example 8-11. Computing Output Offset for Fused Pixel Shuffler 8-21
Example 8-12. Sigmoid Approximation with Minimax Polynomials 8-22
Example 8-13. Sigmoid Approximation with Scalef 8-24
Example 8-14. Pseudocode for Finding Top K 8-26
Example 9-1. Pseudo-code Using CLFLUSH 9-10
Example 9-2. Flushing Cache Lines Using CLFLUSH or CLFLUSHOPT 9-12
Example 9-3. Populating an Array for Circular Pointer Chasing with Constant Stride 9-13
Example 9-4. Prefetch Scheduling Distance 9-16
Example 9-5. Using Prefetch Concatenation 9-17
Example 9-6. Concatenation and Unrolling the Last Iteration of Inner Loop 9-17
Example 9-7. Data Access of a 3D Geometry Engine without Strip-mining 9-21
Example 9-8. Data Access of a 3D Geometry Engine with Strip-mining 9-22
Example 9-9. Using HW Prefetch to Improve Read-Once Memory Traffic 9-23
Example 9-10. Basic Algorithm of a Simple Memory Copy 9-27
Example 9-11. A Memory Copy Routine Using Software Prefetch 9-27
Example 9-12. Memory Copy Using Hardware Prefetch and Bus Segmentation 9-29
Example 10-1. Code Using libnuma to Find the Maximum Number of NUMA Node 10-2
Example 11-1. Serial Execution of Producer and Consumer Work Items 11-5
Example 11-2. Basic Structure of Implementing Producer Consumer Threads 11-6
Example 11-3. Thread Function for an Interlaced Producer Consumer Model 11-7
Example 11-4. Spin-wait Loop and PAUSE Instructions 11-12
Example 11-5. Coding Pitfall using Spin Wait Loop 11-15
Example 11-6. Placement of Synchronization and Regular Variables 11-16
Example 11-7. Declaring Synchronization Variables without Sharing a Cache Line 11-17
Example 11-8. Batched Implementation of the Producer Consumer Threads 11-21
Example 11-9. Parallel Memory Initialization Technique Using OpenMP and NUMA 11-25
Example 13-1. Compute 64-bit Quotient and Remainder with 64-bit Divisor 13-3
Example 13-2. Quotient and Remainder of 128-bit Dividend with 64-bit Divisor 13-4
Example 14-1. A Hash Function Examples 14-4
Example 14-2. Hash Function Using CRC32 14-5
Example 14-3. Strlen() Using General-Purpose Instructions 14-7
Example 14-4. Sub-optimal PCMPISTRI Implementation of EOS handling 14-9
Example 14-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency 14-9
Example 14-6. WordCnt() Using C and Byte-Scanning Technique 14-11
Example 14-7. WordCnt() Using PCMPISTRM 14-12
Example 14-8. KMP Substring Search in C 14-14
Example 14-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic 14-15
Example 14-10. Substring Search Using PCMPISTRI and KMP Overlap Table 14-17
Example 14-11. I Equivalent Strtok_s() Using PCMPISTRI Intrinsic 14-19
Example 14-12. I Equivalent Strupr() Using PCMPISTRM Intrinsic 14-22
Example 14-13. UTF16 VerStrlen() Using C and Table Lookup Technique 14-23
Example 14-14. Assembly Listings of UTF16 VerStrlen() Using PCMPISTRI 14-24
Example 14-15. Intrinsic Listings of UTF16 VerStrlen() Using PCMPISTRI 14-26
Example 14-16. Replacement String Library Strcmp Using Intel® SSE4.2 14-27

EXAMPLES

Example 14-17. High-level flow of Character Subset Validation for String Conversion 14-29
Example 14-18. Intrinsic Listings of atol() Replacement Using PCMPISTRI 14-29
Example 14-19. Auxiliary Routines and Data Constants Used in sse4i_atol() listing 14-32
Example 14-20. Conversion of 64-bit Integer to ASCII 14-34
Example 14-21. Conversion of 64-bit Integer to ASCII without Integer Division 14-35
Example 14-22. Conversion of 64-bit Integer to ASCII Using Intel® SSE4 14-37
Example 14-23. Conversion of 64-bit Integer to Wide Character String Using Intel® SSE4 14-43
Example 14-24. MULX and Carry Chain in Large Integer Numeric 14-48
Example 14-25. Building-block Macro Used in Binary Decimal Floating-point Operations 14-48
Example 15-1. Cartesian Coordinate Transformation with Intrinsics 15-3
Example 15-2. Cartesian Coordinate Transformation with Assembly 15-5
Example 15-3. Direct Polynomial Calculation 15-7
Example 15-4. Function Calls and Intel® AVX/Intel® SSE transitions 15-11
Example 15-5. AoS to SoA Conversion of Complex Numbers in C Code 15-14
Example 15-6. Aos to SoA Conversion of Complex Numbers Using Intel® AVX 15-15
Example 15-7. Register Overlap Method for Median of 3 Numbers 15-17
Example 15-8. Data Gather - Intel® AVX versus Scalar Code 15-19
Example 15-9. Scatter Operation Using Intel® AVX 15-21
Example 15-10. SAXPY using Intel® AVX 15-22
Example 15-11. Using 16-Byte Memory Operations for Unaligned 32-Byte Memory Operation 15-24
Example 15-12. SAXPY Implementations for Unaligned Data Addresses 15-24
Example 15-13. Loop with Conditional Expression 15-28
Example 15-14. Handling Loop Conditional with VMASKMOV 15-28
Example 15-15. Three-Tap Filter in C Code 15-29
Example 15-16. Three-Tap Filter with 128-bit Mixed Integer and FP SIMD 15-29
Example 15-17. 256-bit AVX Three-Tap Filter Code with VSHUFPS 15-30
Example 15-18. Three-Tap Filter Code with Mixed 256-bit AVX and 128-bit AVX Code 15-31
Example 15-19. 8x8 Matrix Transpose - Replace Shuffles with Blends 15-33
Example 15-20. 8x8 Matrix Transpose Using VINSERTPS 15-36
Example 15-21. Port 5 versus Load Port Shuffles 15-38
Example 15-22. Divide Using DIVPS for 24-bit Accuracy 15-41
Example 15-23. Divide Using RCPPS 11-bit Approximation 15-41
Example 15-24. Divide Using RCPPS and Newton-Raphson Iteration 15-41
Example 15-25. Reciprocal Square Root Using DIVPS+SQRTPS for 24-bit Accuracy 15-42
Example 15-26. Reciprocal Square Root Using RSQRTPS 11-bit Approximation 15-43
Example 15-27. Reciprocal Square Root Using RSQRTPS and Newton-Raphson Iteration 15-43
Example 15-28. Square Root Using SQRTPS for 24-bit Accuracy 15-44
Example 15-29. Square Root Using RSQRTPS 11-bit Approximation 15-44
Example 15-30. Square Root Using RSQRTPS and One Taylor Series Expansion 15-45
Example 15-31. Array Sub Sums Algorithm 15-47
Example 15-32. Single-Precision to Half-Precision Conversion 15-48
Example 15-33. Half-Precision to Single-Precision Conversion 15-50
Example 15-34. Performance Comparison of Median3 using Half-Precision vs. Single-Precision 15-51
Example 15-35. FP Mul/FP Add Versus FMA 15-53
Example 15-36. Unrolling to Hide Dependent FP Add Latency 15-53
Example 15-37. FP Mul/FP Add Versus FMA 15-55
Example 15-38. Macros for Separable KLT Intra-block Transformation Using AVX2 15-56
Example 15-39. Separable KLT Intra-block Transformation Using AVX2 15-58
Example 15-40. Macros for Parallel Moduli/Remainder Calculation 15-64
Example 15-41. Signed 64-bit Integer Conversion Utility 15-65
Example 15-42. Unsigned 63-bit Integer Conversion Utility 15-66
Example 15-43. Access Patterns Favoring Non-VGATHER Techniques 15-70
Example 15-44. Access Patterns Likely to Favor VGATHER Techniques 15-71
Example 15-45. Software AVX Sequence Equivalent to Full-Mask VPGATHERD 15-72

EXAMPLES

Example 15-46. AOS to SOA Transformation Alternatives 15-74
Example 15-47. Non-Strided AOS to SOA 15-75
Example 15-48. Conversion to Throughput-Reduced MMX sequence to Intel® AVX2 Alternative 15-77
Example 16-1. Unoptimized Sleep Loop 16-14
Example 16-2. Power Consumption Friendly Sleep Loop Using PAUSE 16-14
Example 17-1. Cartesian Coordinate System Rotation with Intrinsics 17-4
Example 17-2. Cartesian Coordinate System Rotation with Assembly 17-6
Example 17-3. Masking with Intrinsics 17-10
Example 17-4. Masking with Assembly 17-11
Example 17-5. Masking Example 17-12
Example 17-6. Masking vs. Blending Example 1 17-13
Example 17-7. Masking vs. Blending Example 2 17-14
Example 17-8. Multiple Condition Execution 17-15
Example 17-9. Peeling and Remainder Masking 17-17
Example 17-10. Comparing Intel® AVX-512 Data Compress with Alternative 1 17-21
Example 17-11. Comparing Intel® AVX-512 Data Compress with Alternative 2 17-22
Example 17-12. Comparing Intel® AVX-512 Data Compress with Alternative 3 17-23
Example 17-13. Comparing Intel® AVX-512 Data Compress with Alternative 4 17-24
Example 17-14. Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives 17-26
Example 17-15. Comparing Ternary Logic to Alternative 1 17-28
Example 17-16. Comparing Ternary Logic to Alternatives 2 and 3 17-29
Example 17-17. Matrix Transpose Alternatives 17-33
Example 17-18. Broadcast Executed on Load Ports Alternatives 17-35
Example 17-19. 16-bit Broadcast Executed on Port 5 17-35
Example 17-20. Embedded vs Non-Embedded Rounding 17-36
Example 17-21. Scalar Scatter 17-39
Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics 17-41
Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly 17-42
Example 17-24. Scatter Implementation Alternatives 17-53
Example 17-25. Scalar vs. Vector Update Using AVX-512CD 17-55
Example 17-26. Improvement with VPERMB Implementation 17-57
Example 17-27. Improvement with VPERMI2B Implementation 17-60
Example 17-28. Improvement with VPMULTISHIFTQB Implementation 17-63
Example 17-29. 256-bit Code vs. 256-bit Code Mixed with 512-bit Code 17-66
Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture . 17-68
Example 17-31. Gather to Shuffle in Strided Loads Example 17-71
Example 17-32. Gather to Shuffle in Strided Stores Example 17-73
Example 17-33. Gather to Shuffle in Adjacent Loads Example 17-74
Example 17-34. Data Alignment 17-75
Example 17-35. Vectorized 32-bit Float Division, 24 Bits 17-82
Example 17-36. Vectorized 32-bit Float Division, 23 and 14 Bits 17-82
Example 17-37. Reciprocal Square Root, 22 Bits 17-82
Example 17-38. Reciprocal Square Root, 23 and 14 Bits 17-83
Example 17-39. Square Root , 24 Bits 17-83
Example 17-40. Square Root , 23 and 14 Bits 17-83
Example 17-41. Dividing Packed Doubles, 53 and 52 Bits 17-84
Example 17-42. Dividing Packed Doubles, 26 and 14 Bits 17-84
Example 17-43. Reciprocal Square Root of Doubles, 51 Bits 17-85
Example 17-44. Reciprocal Square Root of Doubles, 52 and 50 Bits 17-86
Example 17-45. Reciprocal Square Root of Doubles, 26 and 14 Bits 17-87
Example 17-46. Square Root of Packed Doubles, 53 and 52 Bits 17-87
Example 17-47. Square Root of Packed Doubles, 26 and 14 Bits 17-88
Example 18-1. Converting a Complex-Valued Mask to a Real-Valued Mask by Duplicating Adjacent Bits 18-7

EXAMPLES

Example 18-2. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask By AND-Combining Adjacent Bits 18-8
Example 18-3. Function for Converting from a Real-Valued Mask to a Complex-Valued Mask by OR-Combining Adjacent Bits 18-9
Example 18-4. Function to Implement the 16-Bit Compress Operation on FP16 Vector Elements 18-17
Example 18-5. Function that Performs Fast Floating-Point Minimum Using Integer Instructions 18-19
Example 19-1. Legacy Intel® AES-NI vs. Vector AES 19-1
Example 19-2. SM4 GFNI Encryption Round Example 19-3
Example 20-1. Pseudo-Code for the TILEZERO, TILELOAD, and TILESTORE Instructions 20-6
Example 20-2. B Matrix Re-Layout Procedure 20-6
Example 20-3. Original Layout of 32x16 bfloat16 B-Matrix 20-7
Example 20-4. Re-Layout of 32x16 bfloat16 B-Matrix 20-7
Example 20-5. Original Layout of 64 x 16 unt8 B-Matrix 20-8
Example 20-6. Re-Layout of 64x16 bfloat16 B-Matrix 20-8
Example 20-7. Common Defines 20-9
Example 20-8. Reference GEMM Implementation 20-10
Example 20-9. K-Dimension Loop as Innermost Loop–A, a Highly Inefficient Approach 20-11
Example 20-10. Innermost Loop Tile Pre-Loading 20-12
Example 20-11. Switched Order of M_ACC and N_ACC Loops 20-12
Example 20-12. Optimized GEMM Implementation 20-13
Example 20-13. Dimension of Matrices, Data Types, and Tile Sizes 20-14
Example 20-14. Optimized GEMM Assembly Language Implementation 20-14
Example 20-15. Activations Layout Procedure 20-15
Example 20-16. Weights Re-Layout Procedure 20-16
Example 20-17. Common Defines for Convolution 20-19
Example 20-18. Optimized Direct Convolution Implementation 20-20
Example 20-19. Additional Defines for Convolution with Cache Blocking 20-21
Example 20-20. Optimized Convolution Implementation with Cache Blocking 20-22
Example 20-21. Convolution Code Fused with Post-Convolution Code 20-30
Example 20-22. An Example of a Short GEMM Fused and Pipelined with Quantization and ReLU 20-33
Example 20-23. Two Blocks of 16 Cache Lines of 32-bit Floats Converted to One Block of 16 Cache Lines of 16-bit BFloat 20-37
Example 20-24. Using Unsigned Saturation 20-37
Example 20-25. Prefetching Rows to the DCU 20-41
Example 20-26. BF16 Matrix Transpose (32x8 to 8x32) 20-42
Example 20-27. BF16 VNNI-to-VNNI Transpose (8x8 to 2x32) 20-46
Example 20-28. BF16 Flat-to-VNNI Transpose (16x8 to 4x32) 20-49
Example 20-29. BF16 Flat-to-VNNI Re-Layout 20-51
Example 20-30. GEMM Parallelized with omp Parallel for Collapse 20-55
Example 20-31. Byte Decompression Code with Intel® AVX-512 Intrinsics 20-58
Example 20-32. Identification of Tile Shape Using Parameter m, n, k 20-60
Example 20-33. Intel® AMX Intrinsics Header File 20-60
Example 20-34. Intel® AMX Intrinsics Usage 20-63
Example 20-35. Compiler-Generated Assembly-Level Code from Example 20-30 20-64
Example 20-36. Compiler-Generated Assembly-Level Code Where Tile Register Save/Restore is Optimized Away 20-65

FIGURES

PAGE

Figure 2-1. Layout of Array of Pointers Prefetchers 2-4
Figure 2-2. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture 2-11
Figure 2-3. Processor Front End of the Golden Cove Microarchitecture 2-12
Figure 2-4. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture 2-17
Figure 2-5. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture 2-26
Figure 2-6. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures 2-28
Figure 2-7. Mixed Workloads 2-30
Figure 2-8. LINPACK Performance 2-31
Figure 2-9. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture 2-33
Figure 2-10. Intel® Hyper-Threading Technology on an SMP System 2-41
Figure 2-11. Typical SIMD Operations 2-43
Figure 2-12. SIMD Instruction Register Usage 2-44
Figure 3-1. INT Execution Ports Within the Processor Core Pipeline 3-32
Figure 3-2. Generic Program Flow of Partially Vectorized Code 3-36
Figure 3-3. Memcpy Performance Comparison for Lengths up to 2KB 3-64
Figure 3-4. MariaDB - CHA % Cycles Fast Asserted 3-77
Figure 4-1. Processor Core Pipeline Functionality of the Crestmont Microarchitecture 4-2
Figure 4-2. Front-End Pipeline Functionality of the Crestmont Microarchitecture 4-3
Figure 4-3. Execution Pipeline Functionality of the Crestmont Microarchitecture 4-5
Figure 4-4. Processor Core Pipeline Functionality of the Tremont Microarchitecture 4-12
Figure 5-1. General Procedural Flow of Application Detection of Intel® AVX 5-6
Figure 5-2. General Procedural Flow of Application Detection of Float-16 5-8
Figure 5-3. Converting to Intel® Streaming SIMD Extensions Chart 5-12
Figure 5-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-Offs 5-14
Figure 5-5. Loop Blocking Access Pattern 5-28
Figure 6-1. PACKSSDW mm, mm/mm64 Instruction 6-7
Figure 6-2. Interleaved Pack with Saturation 6-7
Figure 6-3. Result of Non-Interleaved Unpack Low in MM0 6-9
Figure 6-4. Result of Non-Interleaved Unpack High in MM1 6-9
Figure 6-5. PEXTRW Instruction 6-10
Figure 6-6. PINSRW Instruction 6-11
Figure 6-7. PMOVSMKB Instruction 6-13
Figure 6-8. Data Alignment of Loads and Stores in Reverse Memory Copy 6-32
Figure 6-9. A Technique to Avoid Cacheline Split Loads in Reverse Memory Copy Using Two Aligned Loads 6-34
Figure 7-1. Homogeneous Operation on Parallel Data Elements 7-3
Figure 7-2. Horizontal Computation Model 7-3
Figure 7-3. Dot Product Operation 7-4
Figure 7-4. Horizontal Add Using MOVHLPS/MOVLHPS 7-9
Figure 8-1. VPMADDUBSW + VPMADDWD + VPADDD Fused into VPDPBUSD (3x Peak Ops on Server Architectures, 2x Peak Ops on Client Architectures) 8-2
Figure 8-2. Matrix Layout, Inputs and Outputs 8-8
Figure 8-3. Transformed Weights 8-8
Figure 8-4. Convolution Operation 8-9
Figure 8-5. Matrix Multiplications and Summations 8-9
Figure 8-6. 3-Tier Flexible 2D Blocking 8-10
Figure 8-7. 3-Tier Flexible 2D Blocking Loops 8-11
Figure 8-8. Standard vs Optimized vs. Low OFM Optimized Data Layouts 8-14
Figure 8-9. Dynamic Batch Size 8-25
Figure 8-10. Find Top 16 Values in Some Input 8-25
Figure 9-1. CLFLUSHOPT versus CLFLUSH In Sky Lake Microarchitecture 9-11
Figure 9-2. Effective Latency Reduction as a Function of Access Stride 9-14
Figure 9-3. Memory Access Latency and Execution Without Prefetch 9-14
Figure 9-4. Memory Access Latency and Execution With Prefetch 9-15

FIGURES

PAGE

Figure 9-5. Prefetch and Loop Unrolling 9-18
Figure 9-6. Memory Access Latency and Execution With Prefetch 9-18
Figure 9-7. Spread Prefetch Instructions 9-19
Figure 9-8. Cache Blocking – Temporally Adjacent and Non-adjacent Passes 9-20
Figure 9-9. Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops 9-21
Figure 9-10. Single-Pass vs. Multi-Pass 3D Geometry Engines 9-25
Figure 10-1. Example of SNC Configuration 10-1
Figure 10-2. NUMA Disabled 10-5
Figure 10-3. SNC Off 10-6
Figure 10-4. SNC On 10-7
Figure 10-5. Domain Example with One MPI Process Per Domain 10-8
Figure 11-1. Amdahl's Law and MP Speed-up 11-2
Figure 11-2. Single-threaded Execution of Producer-consumer Threading Model 11-5
Figure 11-3. Execution of Producer-consumer Threading Model on a Multicore Processor 11-5
Figure 11-4. Interlaced Variation of the Producer Consumer Model 11-7
Figure 11-5. Batched Approach of Producer Consumer Model 11-21
Figure 12-1. In App Direct Mode, Data on the Intel® Optane™ DC Persistent Memory Module is Accessed Directly with Loads and Stores 12-2
Figure 12-2. Decision Flow for Determining When to Use Intel® Optane™ DC Persistent Memory Module vs. DRAM 12-3
Figure 12-3. Loaded Latency Curves for One Intel® Optane™ DC Persistent Memory Module DIMM: Sequential Traffic (Left) and Random Traffic (Right) 12-5
Figure 12-4. Number of Threads vs. Bandwidth 12-6
Figure 12-5. Combining with Two Cores 12-7
Figure 12-6. Combining with Four Cores 12-7
Figure 12-7. Combining with Eight Cores 12-8
Figure 12-8. PMDK vs. MSYNC Flushing Times 12-10
Figure 12-9. Bandwidth vs. Power Consumption 12-10
Figure 12-10. Read-Write Equivalence for Intel® Optane™ DC Persistent Memory Module DIMMs within Different Power Budgets 12-11
Figure 12-11. Bandwidth Available to Software when There is No Locality at 256B Granularity 12-12
Figure 14-1. Intel® SSE4.2 String/Text Instruction Immediate Operand Control 14-2
Figure 14-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search 14-13
Figure 14-3. Intel® SSE4.2 Speedup of SubString Searches 14-19
Figure 14-4. Compute Four Remainders of Unsigned Short Integer in Parallel 14-37
Figure 15-1. Intel® AVX—Intel® SSE Transitions in Earlier Microarchitectures 15-9
Figure 15-2. Intel® AVX- Intel® SSE Transitions in the Skylake Microarchitecture 15-9
Figure 15-3. Source Location of Each Destination Element in a 128-bit Lane 15-12
Figure 15-4. Gather Operation 15-18
Figure 15-5. Using VPERM2F128 to Swap Elements Between Lanes 15-35
Figure 15-6. Step One of 8 x 8 Matrix Transpose Using VINSERTF128 15-35
Figure 15-7. Intel AVX Implementation of the Complex Multiply Algorithm 15-37
Figure 15-8. SSE Implementation Description 15-46
Figure 15-9. Intel® AVX Implementation of the Array Sub Sums Algorithm 15-47
Figure 15-10. 4x4 Image Block Transformation 15-56
Figure 15-11. Throughput Comparison of Gather Instructions 15-72
Figure 15-12. Comparison of HW GATHER Versus Software Sequence in Skylake Microarchitecture 15-73
Figure 16-1. Performance History and State Transitions 16-2
Figure 16-2. Active Time Versus Halted Time of a Processor 16-3
Figure 16-3. Application of C-States to Idle Time 16-4
Figure 16-4. Profiles of Coarse Task Scheduling and Power Consumption 16-9
Figure 16-5. Thread Migration in a Multicore Processor 16-11

FIGURES

PAGE

Figure 16-6. Progression to Deeper Sleep 16-11
Figure 16-7. Energy Savings Due to Performance Optimization 16-13
Figure 16-8. Energy Savings Due to Vectorization 16-13
Figure 16-9. Energy-Saving Comparison of Synchronization Primitives 16-16
Figure 16-10. Power-Saving Comparison of Power-Source-Aware Frame Rate Configurations 16-17
Figure 17-1. Cartesian Rotation 17-3
Figure 17-2. Mask Move When Using Merging Masking 17-9
Figure 17-3. Mask Move Operation When Using Merging Masking 17-9
Figure 17-4. Result of Execution with Zeroing Masking 17-9
Figure 17-5. Data Forwarding Cases 17-19
Figure 17-6. Data Compress Operation 17-20
Figure 17-7. Data Expand Operation 17-25
Figure 17-8. Ternary Logic Example 1 Truth Table 17-27
Figure 17-9. Ternary Logic Example 2 Truth Table 17-30
Figure 17-10. VPERMI2PS Instruction Operation 17-31
Figure 17-11. Two-Source Permute Instructions in a Matrix Transpose Operation 17-32
Figure 17-12. VSCATTERDPD Instruction Operation 17-38
Figure 17-13. VPCONFLICTD Instruction Execution 17-51
Figure 17-14. VPCONFLICTD Merging Process 17-51
Figure 17-15. VPCONFLICTD Permute Control 17-52
Figure 17-16. VPCONFLICTD ZMM2 Result 17-54
Figure 17-17. Sparse Vector Example 17-54
Figure 17-18. VPERMB Instruction Operation 17-56
Figure 17-19. VPERMI2B Instruction Operation 17-58
Figure 17-20. VPERMT2B Instruction Operation 17-59
Figure 17-21. VPMULTISHIFTQB Instruction Operation 17-61
Figure 17-22. Fast Bypass When All Sources Come from FMA Unit 17-64
Figure 17-23. Mixing Intel AVX Instructions or Intel AVX-512 Instructions with Intel SSE Instructions 17-65
Figure 18-1. Layout of a 128-Bit Register Representing Four Complex FP16 (CFP16) Values 18-3
Figure 18-2. A Zero-Masked FP16 Add On Two 128-Bit Vectors 18-5
Figure 18-3. A Masked Complex Multiplication 18-6
Figure 18-4. Using a Real-Valued FP16 Vector Operation for Implementing a Masked Complex Addition . . 18-7
Figure 18-5. Comparison Operation Between Two Complex-Valued Vectors 18-8
Figure 18-6. Bit Layout of Three Types of Floating-Point Formats 18-10
Figure 18-7. Landmark Numbers on the Real-Valued FP16 Axis 18-10
Figure 18-8. Heat-map Showing Relative ULP Error for Different Combinations of Divisor and Dividend Value Ranges 18-16
Figure 20-1. Matrix Notation 20-4
Figure 20-2. Intel® AMX Multiplication with Max-sized int8 Tiles 20-5
Figure 20-3. Activations layout 20-16
Figure 20-4. Weights Re-Layout 20-17
Figure 20-5. Convolution-Matrix Multiplication and Summation Equivalence 20-17
Figure 20-6. Matrix-Like Multiplications Part of a Convolution 20-18
Figure 20-7. Batching Execution Using Six Layers with Four Instances Per Thread 20-25
Figure 20-8. A Convolution Example 20-28
Figure 20-9. A Convolution Example with Large Tiles 20-28
Figure 20-10. Using TILEZERO to Solve Performance Degradation 20-35
Figure 20-11. A Conversion Flow of 32-bit Integers to 8-bit Integers 20-36
Figure 20-12. Trivial Deep Learning Topology with Naive Buffer Allocation 20-38
Figure 20-13. Minimal Memory Footprint Buffer Allocation Scheme for Trivial Deep Learning Topology . . . 20-38
Figure 20-14. Flat-to-VNNI Transpose of WORDs Equivalence to Flat-to-Flat Transpose of DWORDs 20-48
Figure 20-15. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the M-Dimension . . 20-53
Figure 20-16. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the N-Dimension . . 20-54
Figure 20-17. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the K-Dimension . . . 20-54

FIGURES

PAGE

Figure 20-18. A Recommendation System Multi-Threading Model 20-57
Figure 20-19. Data Expand Operation 20-58
Figure 22-1. General TMA Hierarchy for Out-of-Order Microarchitectures 22-2
Figure 22-2. TMA's Top Level Drill Down Flowchart 22-3
Figure 22-3. TMA Hierarchy and Precise Events in the Skylake Microarchitecture 22-8
Figure 22-4. System Topology Supported by Intel® Xeon® Processor 5500 Series 22-16
Figure 22-5. PMU Specific Event Logic Within the Pipeline 22-18
Figure 22-6. LBR Records and Basic Blocks 22-29
Figure 22-7. Using LBR Records to Rectify Skewed Sample Distribution 22-30
Figure 22-8. RdData Request after LLC Miss to Local Home (Clean Rsp) 22-41
Figure 22-9. RdData Request after LLC Miss to Remote Home (Clean Rsp) 22-41
Figure 22-11. RdData Request after LLC Miss to Local Home (Hitm Response) 22-42
Figure 22-10. RdData Request after LLC Miss to Remote Home (Hitm Response) 22-42
Figure 22-12. RdData Request after LLC Miss to Local Home (Hit Response) 22-43
Figure 22-13. RdInvOwn Request after LLC Miss to Remote Home (Clean Res) 22-43
Figure 22-15. RdInvOwn Request after LLC Miss to Local Home (Hit Res) 22-44
Figure 22-14. RdInvOwn Request after LLC Miss to Remote Home (Hitm Res) 22-44
Figure 22-16. Performance Events Drill-Down and Software Tuning Feedback Loop 22-65
Figure B-1. ITLB and ITLB 4K MPKI Across Runtime Workloads B-4
Figure B-2. measure-perf-metric.sh Tool Usage for Process ID 69772 for 30 Seconds B-6
Figure B-3. Using measure-perf-metric.sh with -r to Determine Where TLB Misses are Coming From B-6
Figure B-4. Commands for Checking Linux* Distribution for THP B-7
Figure B-5. API Calls Provided by the Intel Reference Implementation B-9
Figure B-6. perf Output Will Not Have the Proper Symbols After Large Page Mapping B-11
Figure B-7. Using Perf Record with -e frontend_retired.itlb_miss to Determine ITLB Misses B-15
Figure B-8. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload B-15
Figure B-9. Using FlameScope to Visualize the ITLB Misses Heatmap from the WebTooling Workload B-16
Figure B-10. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload B-16
Figure B-11. Visualizing ITLB Miss Trends for "Built-in" Functions from the Ghost.js Workload When Run With Large Pages B-17

Numerics

64-bit mode
arithmetic, 13-5
coding guidelines, 13-1
compiler settings, A-2
CVTSI2SD instruction, 13-6
CVTSI2SS instruction, 13-6
default operand size, 13-1
legacy instructions, 13-1
multiplication notes, 13-2
register usage, 13-1, 13-5
REX prefix, 13-1
sign-extension, 13-5
software prefetch, 13-6

A

absolute difference of signed numbers, 6-18
absolute difference of unsigned numbers, 6-17
absolute value, 6-18
active power, 16-1
algorithm to avoid changing the rounding mode, 3-69
alignment
arrays, 3-50
code, 3-8
stack, 3-52
structures, 3-50
Amdahl's law, 11-1
AoS format, 5-24
application performance tools, A-1
arrays
aligning, 3-50
automatic vectorization, 5-17

B

battery life
guidelines for extending, 16-6
mobile optimization, 16-1
OS APIs, 16-7
quality trade-offs, 16-6
branch prediction
choosing types, 3-8
code examples, 3-5
eliminating branches, 3-4
optimizing, 3-4
unrolling loops, 3-10

C

C4-state, 16-4
cache management
blocking techniques, 9-21
cache level, 9-3
CLFLUSH instruction, 9-9
coding guidelines, 8-3, 8-5, 8-12, 8-14, 8-16, 8-18, 8-19, 8-20, 8-21, 8-22, 8-24, 8-26, 9-1, 15-3, 17-4
CPUID instruction, 3-3, 9-32
function leaf, 3-3
optimizing, 9-1
simple memory copy, 9-28
video decoder, 9-27
video encoder, 9-27
See also: optimizing cache utilization
CD/DVD, 16-7
changing the rounding mode, 3-69
classes (C/C++), 5-16
CLFLUSH instruction, 9-9
clipping to an arbitrary signed range, 6-22
clipping to an arbitrary unsigned range, 6-24
coding techniques, 5-14, 11-19
64-bit guidelines, 13-1
absolute difference of signed numbers, 6-18
absolute difference of unsigned numbers, 6-17
absolute value, 6-18
clipping to an arbitrary signed range, 6-22
clipping to an arbitrary unsigned range, 6-24
conserving power, 16-7
data in segment, 3-53
generating constants, 6-16
interleaved pack with saturation, 6-7
interleaved pack without saturation, 6-8
methodologies, 5-14
non-interleaved unpack, 6-9
optimization options, A-2
rules, 3-3
signed unpack, 6-6
simplified clip to arbitrary signed range, 6-23
sleep transitions, 16-8
suggestions, 3-3
tuning hints, 3-3
unsigned unpack, 6-6
See also: floating-point code
coherent requests, 9-7
command-line options
inline expansion of library functions, A-3
vectorizer switch, A-3
comparing register values, 3-25, 3-27
compatibility mode, 13-1
compiler intrinsics
_mm_load, 9-27
_mm_prefetch, 9-27
_mm_stream, 9-27
compilers
documentation, 1-7
general recommendations, 3-2
plug-ins, A-2
supported alignment options, 5-21
See also: Intel C++ Compiler & Intel Fortran Compiler
computation
intensive code, 5-13
converting 64-bit to 128-bit SIMD integers, 6-36
converting code to MMX technology, 5-11
CPUID instruction
cache paramaters, 9-32
function leaf, 9-32
function leaf 4, 3-3
Intel compilers, 3-2
MMX support, 5-2
SSE support, 5-2
SSE2 support, 5-2
SSE3 support, 5-3
SSSE3 support, 5-3, 5-4
strategy for use, 3-2
C-states, 16-1, 16-3
CVTSI2SD instruction, 13-6
CVTSI2SS instruction, 13-6
CVTTPS2PI instruction, 7-11
CVTTSS2SI instruction, 7-11

D

data
access pattern of array, 3-51
aligning arrays, 3-50
aligning structures, 3-50
alignment, 5-19
arrangement, 7-2
code segment and, 3-53
deswizzling, 7-7
swizzling, 7-5
swizzling using intrinsics, 7-6
deeper sleep, 16-4
denormals-are-zero (DAZ), 7-11
deterministic cache parameters
cache sharing, 9-32, 9-33
multicore, 9-33
overview, 9-32
prefetch stride, 9-33
domain decomposition, 11-4

INDEX

E

eliminating branches, 3-6, 16-14, 16-15
EMMS instruction, 6-2, 6-3
  guidelines for using, 6-2
Enhanced Intel SpeedStep Technology
  description of, 16-8
  multicore processors, 16-10
  usage scenario, 16-1
extract word instruction, 6-10

F

fencing operations, 9-6
  LFENCE instruction, 9-9
  MFENCE instruction, 9-9
FIST instruction, 3-69
FLDCW instruction, 3-69
floating-point code
  arithmetic precision options, A-3
  data arrangement, 7-2
  data deswizzling, 7-7
  data swizzling using intrinsics, 7-6
  guidelines for optimizing, 3-66
  horizontal ADD, 7-9
  improving parallelism, 3-70
  memory access stall information, 3-47
  operations, integer operands, 3-71
  optimizing, 3-66
  planning considerations, 7-1, 8-1
  rules and suggestions, 7-1, 8-1
  scalar code, 7-2
  transcendental functions, 3-71
  unrolling loops, 3-10
  vertical versus horizontal computation, 7-3
  See also: coding techniques
flush-to-zero (FTZ), 7-11
front end
  branching ratios, 22-70
  characterizing mispredictions, 22-71
  loop unrolling, 11-24
  optimization, 3-4
functional decomposition, 11-4
FXCH instruction, 3-70, 7-2

G

generating constants, 6-16
GetActivePwrScheme, 16-7
GetSystemPowerStatus, 16-7

H

hardware multithreading
  support for, 3-3
hardware prefetch
  cache blocking techniques, 9-24
  latency reduction, 9-13
  memory optimization, 9-12
  operation, 9-12
horizontal computations, 7-9
hotspots
  definition of, 5-13
  identifying, 5-13
  VTune analyzer, 5-13
Hyper-Threading Technology
  avoid excessive software prefetches, 11-20
  bus optimization, 11-10
  cache blocking technique, 11-21
  conserve bus command bandwidth, 11-19
  eliminate 64-K-aliased data accesses, 11-24
  excessive loop unrolling, 11-24
  front-end optimization, 11-24
  full write transactions, 11-21
  functional decomposition, 11-4
  improve effective latency of cache misses, 11-20
  memory optimization, 11-21
  minimize data sharing between physical processors, 11-22
  multitasking environment, 11-3
  optimization, 11-1
  optimization guidelines, 11-9
  optimization with spin-locks, 11-15
  overview, 2-40
  parallel programming models, 11-4
  pipeline, 2-42
  placement of shared synchronization variable, 11-17
  prevent false-sharing of data, 11-17
  processor resources, 2-41
  shared execution resources, 11-28
  shared-memory optimization, 11-22
  synchronization for longer periods, 11-15
  synchronization for short periods, 11-13
  system bus optimization, 11-19
  thread sync practices, 11-10
  thread synchronization, 11-11
  tools for creating multithreaded applications, 11-9

I

IA32_PERFEVSELx MSR, 22-69
increasing bandwidth
  memory fills, 6-32
  video fills, 6-32
Indefinite
  description of, 2-47, 2-48
indirect branch, 3-8
inline assembly, 6-3
inline expansion library functions option, A-3
inlined-asm, 5-15
insert word instruction, 6-11
instruction scheduling, 3-53
Intel 64 and IA-32 processors, 2-1
Intel Core Duo processors
  128-bit integers, 6-37
  performance events, 22-63
  special programming models, 11-5
  static prediction, 3-6
Intel Core microarchitecture
  event ratios, 22-69
  special programming models, 11-5
  static prediction, 3-7
Intel Core Solo processors
  128-bit SIMD integers, 6-37
  performance events, 22-63
  static prediction, 3-6
Intel C++ Compiler, 3-1
  64-bit mode settings, A-2
  description, A-1
  IA-32 settings, A-2
  multithreading support, A-3
  OpenMP, A-3
  optimization settings, A-2
  related Information, 1-6
Intel Debugger
  description, A-1
Intel Enhanced Deeper Sleep
  C-state numbers, 16-3
  enabling, 16-9
  multiple-cores, 16-12
Intel Fortran Compiler
  description, A-1
  multithreading support, A-3
  OpenMP, A-3
  optimization settings, A-2
  related information, 1-6
Intel Integrated Performance Primitives
  for Linux, A-5
  for Windows, A-5
Intel Math Kernel Library for Linux, A-5
Intel Math Kernel Library for Windows, A-5
Intel Mobile Platform SDK, 16-7
Intel NetBurst microarchitecture
  introduction, 2-40
Intel Performance Libraries

benefits, A-5
optimizations, A-5
Intel performance libraries
description, A-1
Intel Performance Tools, 3-1
Intel Thread Checker
example output, A-5, A-7, A-8
Intel Threading Tools, A-7
Intel VTune Performance Analyzer
code coach, 5-13
coverage, 3-2
interleaved pack with saturation, 6-7
interleaved pack without saturation, 6-8
interprocedural optimization, A-4
introduction
optimization features, 2-1
processors covered, 1-1
references, 1-7
IPO. See interprocedural optimization

L

large load stalls, 3-48
latency, 9-2, 9-14
legacy mode, 13-1
LFENCE Instruction, 9-9
links to web data, 1-7
load instructions and prefetch, 9-5
loading-storing to-from same DRAM page, 6-33
loop
blocking, 5-27
unrolling, 9-18, A-3

M

MASKMOVDQU instruction, 9-6
memory bank conflicts, 9-2
memory optimizations
loading-storing to-from same DRAM page, 6-33
overview, 6-31
partial memory accesses, 6-31, 6-33
performance, 5-23
reference instructions, 3-24
using aligned stores, 6-33
using prefetch, 9-12
MFENCE instruction, 9-9
misaligned data access, 5-19
misalignment in the FIR filter, 5-20
mobile computing
ACPI standard, 16-1, 16-3
active power, 16-1
battery life, 16-1, 16-6, 16-7
C4-state, 16-4
CD/DVD, WLAN, WiFi, 16-7
C-states, 16-1, 16-3
deep sleep transitions, 16-8
deeper sleep, 16-4, 16-9
Intel Mobile Platform SDK, 16-7
OS APIs, 16-7
OS changes processor frequency, 16-2
OS synchronization APIs, 16-7
overview, 16-1
performance options, 16-6
platform optimizations, 16-7
P-states, 16-1
Speedstep technology, 16-8
spin-loops, 16-7
state transitions, 16-2
static power, 16-1
WM_POWERBROADCAST message, 16-8
MOVAPD instruction, 7-2
MOVAPS instruction, 7-2
move byte mask to integer, 6-14
MOVHLPS instruction, 7-9
MOVLHPS instruction, 7-9
MOVNTDQ instruction, 9-6
MOVNTI instruction, 9-6
MOVNTPD instruction, 9-6
MOVNTPS instruction, 9-6
MOVNTQ instruction, 9-6
MOVQ Instruction, 6-33
MOVUPD instruction, 7-2
MOVUPS instruction, 7-2
multicore processors
architecture, 2-1
C-state considerations, 16-11
energy considerations, 16-10
SpeedStep technology, 16-10
thread migration, 16-11
multiprocessor systems
dual-core processors, 11-1
HT Technology, 11-1
optimization techniques, 11-1
See also: multithreading & Hyper-Threading Technology
multithreading
Amdahl's law, 11-1
application tools, 11-9
bus optimization, 11-10
compiler support, A-3
dual-core technology, 3-3
environment description, 11-1
guidelines, 11-9
hardware support, 3-3
HT technology, 3-3
Intel Core microarchitecture, 11-5
parallel & sequential tasks, 11-1
programming models, 11-3
shared execution resources, 11-28
specialized models, 11-5
thread sync practices, 11-10
See Hyper-Threading Technology

N

Newton-Raphson iteration, 7-1
non-coherent requests, 9-8
non-interleaved unpack, 6-9
non-temporal stores, 9-6, 9-26
NOP, 3-27

O

OpenMP compiler directives, 11-9, A-3
optimization
branch prediction, 3-4
branch type selection, 3-8
eliminating branches, 3-4
features, 2-1
general techniques, 3-1
static prediction, 3-6
unrolling loops, 3-10
optimizing cache utilization
cache management, 9-27
examples, 9-9
non-temporal store instructions, 9-6, 9-8
prefetch and load, 9-5
prefetch instructions, 9-3
prefetching, 9-3
SFENCE instruction, 9-9
streaming, non-temporal stores, 9-6
See also: cache management
OS APIs, 16-7

P

pack instructions, 6-7
packed average byte or word), 6-25
packed multiply high unsigned, 6-24
packed shuffle word, 6-14
packed signed integer word maximum, 6-24
packed sum of absolute differences, 6-25
parallelism, 5-14, 11-4

partial memory accesses, 6-31
PAUSE instruction, 11-10
PAVGB instruction, 6-25
PAVGW instruction, 6-25
PeekMessage(), 16-7
Pentium 4 processors
  static prediction, 3-6
Pentium M processors
  static prediction, 3-6
performance models
  Amdahl's law, 11-1
  multithreading, 11-1
  parallelism, 11-1
  usage, 11-1
performance monitoring events
  analysis techniques, 22-64
  Bus_Not_In_Use, 22-64
  Bus_Snoops, 22-64
  DCU_Snoop_to_Share, 22-64
  drill-down techniques, 22-64
  event ratios, 22-69
  Intel Core Duo processors, 22-63
  Intel Core Solo processors, 22-63
  Intel Netburst architecture, 22-1
  Intel Xeon processors, 22-1
  L1_Pref_Req, 22-64
  L2_No_Request_Cycles, 22-64
  L2_Reject_Cycles, 22-64
  Pentium 4 processor, 22-1
  performance counter, 22-63
  ratio interpretation, 22-63
  See also: clock ticks
  Serial_Execution_Cycles, 22-64
  Unhalted_Core_Cycles, 22-64
  Unhalted_Ref_Cycles, 22-64
performance tools, 3-1
PEXTRW instruction, 6-10
PGO. See profile-guided optimization
PINSRW instruction, 6-11
PMINSW instruction, 6-24
PMINUB instruction, 6-24
PMOVMSKB instruction, 6-14
PMULHUW instruction, 6-24, 6-25
predictable memory access patterns, 9-3
prefetch
  64-bit mode, 13-6
  coding guidelines, 9-1
  concatenation, 9-17
  deterministic cache parameters, 9-32
  hardware mechanism, 9-2
    characteristics, 9-12
    latency, 9-13
  how instructions designed, 9-3
  innermost loops, 9-3
  instruction considerations
    cache block techniques, 9-21
    checklist, 9-16
    concatenation, 9-17
    hint mechanism, 9-3
    minimizing number, 9-18
    scheduling distance, 9-16
    single-pass execution, 9-2, 9-25
    spread with computations, 9-20
    strip-mining, 9-22
    summary of, 9-2
  instruction variants, 9-3
  latency hiding/reduction, 9-14
  load Instructions, 9-5
  memory access patterns, 9-3
  memory optimization with, 9-12
  minimizing number of, 9-18
  scheduling distance, 9-2, 9-16
  software data, 9-3
  spreading, 9-20
  when introduced, 9-1
PREFETCHNTA instruction, 9-22
  usage guideline, 9-2
PREFETCHT0 instruction, 9-22
  usage guideline, 9-2
producer-consumer model, 11-5
profile-guided optimization, A-4
PSADBW instruction, 6-25
PSHUF instruction, 6-14
P-states, 16-1

Q

-Qparallel, 11-9

R

ratios, 22-69
  branching and front end, 22-70
references, 1-7
releases of, 2-44

S

sampling
  event-based, A-6
scheduling distance (PSD), 9-16
Self-modifying code, 3-53
SFENCE Instruction, 9-9
SHUFPS instruction, 7-3
signed unpack, 6-6
SIMD
  auto-vectorization, 5-17
  cache instructions, 9-1
  classes, 5-16
  coding techniques, 5-14
  data alignment for 128-bits, 5-21
  data alignment for MMX, 5-20
  data and stack alignment, 5-19
  example computation, 2-42
  history, 2-42
  identifying hotspots, 5-13
  instruction selection, 5-30
  loop blocking, 5-27
  memory utilization, 5-23
  MMX technology support, 5-2
  padding to align data, 5-19
  parallelism, 5-14
  SSE support, 5-2
  SSE2 support, 5-2
  SSE3 support, 5-3
  SSSE3 support, 5-3, 5-4
  stack alignment for 128-bits, 5-20
  strip-mining, 5-26
  using arrays, 5-19
  vectorization, 5-14
  VTune capabilities, 5-13
SIMD floating-point instructions
  data arrangement, 7-2
  data deswizzling, 7-7
  data swizzling, 7-5
  general rules, 7-1
  horizontal ADD, 7-9
  planning considerations, 7-1, 8-1
  reciprocal instructions, 7-1
  scalar code, 7-2
  using
    CVTTPS2PI, 7-11
    CVTTSS2SI, 7-11
    FXCH, 7-2
    MOVAPD, 7-2
    MOVAPS, 7-2
    MOVHLPS, 7-9
    MOVLHPS, 7-9
    MOVUPD, 7-2
    MOVUPS, 7-2
    SHUFPS, 7-3
  vertical vs horizontal computation, 7-3

with x87 FP instructions, 7-2
SIMD technology, 2-44
SIMD-integer instructions
64-bits to 128-bits, 6-36
data alignment, 6-4
data movement techniques, 6-5
extract word, 6-10
integer intensive, 6-1
memory optimizations, 6-31
move byte mask to integer, 6-14
packed average byte or word), 6-25
packed multiply high unsigned, 6-24
packed shuffle word, 6-14
packed signed integer word maximum, 6-24
packed sum of absolute differences, 6-25
rules, 6-1
signed unpack, 6-6
unsigned unpack, 6-6
using
EMMS, 6-2
MOVDQ, 6-33
MOVQ2DQ, 6-16
PABSW, 6-18
PACKSSDW, 6-7
PADDQ, 6-26
PALIGNR, 6-4
PAVGB, 6-25
PAVGW, 6-25
PEXTRW, 6-10
PINSRW, 6-11
PMADDWD, 6-26
PMAXSW, 6-24
PMAXUB, 6-24
PMINSW, 6-24
PMINUB, 6-24
PMOVMSKB, 6-14
PMULHUW, 6-24, 6-25
PMULHW, 6-24, 6-25
PMULUDQ, 6-25
PSADBW, 6-25
PSHUF, 6-14
PSHUFB, 6-19, 6-21
PSHUFLW, 6-15
PSLLDQ, 6-26
PSRLDQ, 6-26
PSUBQ, 6-26
PUNPCHQDQ, 6-16
PUNPCKLQDQ, 6-16
simplified 3D geometry pipeline, 9-14
simplified clipping to an arbitrary signed range, 6-23
single vs multi-pass execution, 9-25
sleep transitions, 16-8
SoA format, 5-24
software write-combining, 9-26
spin-loops, 16-7
related information, 1-7
SSE, 2-45
SSE2, 2-45
SSE3, 2-45
SSSE3, 2-45, 2-46
stack
alignment 128-bit SIMD, 5-20
alignment stack, 3-52
dynamic alignment, 3-52
state transitions, 16-2
static power, 16-1
static prediction, 3-6
streaming stores, 9-6
coherent requests, 9-7
improving performance, 9-6
non-coherent requests, 9-8
strip-mining, 5-26, 5-27, 9-22, 9-23
prefetch considerations, 9-24
structures
aligning, 3-50
system bus optimization, 11-19

T

time-consuming innermost loops, 9-3
transaction lookaside buffer, 9-28
transcendental functions, 3-71

U

unpack instructions, 6-9
unrolling loops
benefits of, 3-10
code examples, 3-11
unsigned unpack, 6-5
using MMX code for copy, shuffling, 7-9

V

vector class library, 5-17
vectorized code
auto generation, A-4
automatic vectorization, 5-17
high-level examples, A-4
parallelism, 5-14
SIMD architecture, 5-14
switch options, A-3
vertical vs horizontal computation, 7-3

W

WaitForSingleObject(), 16-7
WaitMessage(), 16-7
weakly ordered stores, 9-6
WiFi, 16-7
WLAN, 16-7
write-combining
buffer, 9-27
memory, 9-27
semantics, 9-7

X

XCHG EAX,EAX, support for, 3-28
XFEATURE_ENALBED_MASK, 5-5
XRSTOR, 5-5
XSAVE, 5-5, 5-8, 5-9, 5-10

Z

, A-1

INDEX

USER RULES

User/Source Coding Rule 1. (M impact, L generality) If an indirect branch has two or more common taken targets and at least one of those targets is correlated with branch history leading up to the branch, then convert the indirect branch to a tree where one or more indirect branches are preceded by conditional branches to those targets. Apply this "peeling" procedure to the common target of an indirect branch that correlates to branch history 3-9

User/Source Coding Rule 2. (H impact, M generality) Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible 3-37

User/Source Coding Rule 3. (M impact, ML generality) Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence 3-37

User/Source Coding Rule 4. (M impact, ML generality) Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches 3-37

User/Source Coding Rule 5. (M impact, ML generality) Keep induction (loop) variable expressions simple . 3-37

User/Source Coding Rule 6. (H impact, M generality) Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary 3-52

User/Source Coding Rule 7. (M impact, L generality) Beware of false sharing within a cache line (64 bytes) 3-54

User/Source Coding Rule 8. (H impact, H generality) Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops 3-57

User/Source Coding Rule 9. (M impact, M generality) Enable the compiler's use of Intel SSE, Intel SSE2, Intel AVX, Intel AVX2, and possibly more advanced SIMD instruction sets (Intel AVX-512) with appropriate switches. Favor scalar SIMD code generation to replace x87 code generation 3-69

User/Source Coding Rule 10. (H impact, ML generality) Make sure your application stays in range to avoid denormal values, underflows 3-69

User/Source Coding Rule 11. (M impact, ML generality) Usually, math libraries take advantage of the transcendental instructions (for example, FSIN) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider an alternate, software-based approach, such as a look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision and the size of the look-up table, and by taking advantage of the parallelism of the Intel SSE and the Intel SSE2 instructions. 3-69

User/Source Coding Rule 12. (H impact, ML generality) Denormalized floating-point constants should be avoided as much as possible 3-70

User/Source Coding Rule 13. If CLFLUSHOPT is available, use CLFLUSHOPT over CLFLUSH and use SFENCE to guard CLFLUSHOPT to ensure write order is globally observed. If CLUSHOPT is unavailable, consider flushing large buffers with CLFLUSH in smaller chunks of less than 4KB. 9-11

User/Source Coding Rule 14. (M impact, H generality) Insert the PAUSE instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance. 11-14

User/Source Coding Rule 15. (M impact, L generality) Replace a spin lock that may be acquired by multiple threads with pipelined locks so no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to use a lock. 11-15

User/Source Coding Rule 16. (H impact, M generality) Use a thread-blocking API in a long idle loop to free up the processor 11-15

User/Source Coding Rule 17. (H impact, M generality) Beware of false sharing within a cache line or within a sector. Allocate critical data or locks separately using alignment granularity not smaller than the "false-sharing threshold" 11-17

USER RULES

User/Source Coding Rule 18. (M impact, ML generality) Place each synchronization variable alone, separated by 128 bytes or in a separate cache line. 11-18

User/Source Coding Rule 19. (H impact, L generality) Do not place any spin lock variable to span a cache line boundary 11-18

User/Source Coding Rule 20. (M impact, H generality) Improve data and code locality to conserve bus command bandwidth. 11-19

User/Source Coding Rule 21. (M impact, L generality) Avoid excessive use of software prefetch instructions and allow automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately. 11-20

User/Source Coding Rule 22. (M impact, M generality) Consider using overlapping multiple back-to-back memory reads to improve effective cache miss latencies. 11-20

User/Source Coding Rule 23. (M impact, M generality) Consider adjusting the sequencing of memory references so the distribution of distances of successive cache misses of the last level cache peaks towards 64 bytes. 11-21

User/Source Coding Rule 24. (M impact, M generality) Use full write transactions to achieve higher data throughput. 11-21

User/Source Coding Rule 25. (H impact, H generality) Use cache blocking to improve locality of data access. Target one quarter to one half of the cache size when targeting Intel processors supporting HT Technology or target a block size that allow all the logical processors serviced by a cache to share that cache simultaneously. 11-22

User/Source Coding Rule 26. (H impact, M generality) Minimize the sharing of data between threads that execute on different bus agents sharing a common bus. The situation of a platform consisting of multiple bus domains should also minimize data sharing across bus domains 11-22

User/Source Coding Rule 27. (H impact, H generality) Minimize data access patterns that are offset by multiples of 64 KBytes in each thread. 11-24

User/Source Coding Rule 28. (M impact, L generality) Avoid excessive loop unrolling to ensure the LSD is operating efficiently 11-24

User/Source Coding Rule 29. Factor in precision and rounding characteristics of FMA instructions when replacing multiply/add operations executing non-FMA instructions. 15-53

User/Source Coding Rule 30. Factor in result-dependency, latency of FP add vs. FMA instructions when replacing FP add operations with FMA instructions 15-53

User/Source Coding Rule 31. Consider using unrolling technique for loops containing back-to-back dependent FMA, FP Add or Vector MUL operations, The unrolling factor can be chosen by considering the latency of the critical instruction of the dependency chain and the number of pipes available to execute that instruction 15-55

USER RULES

Assembler/Compiler Coding Rule 1. (MH impact, M generality) Arrange code to make basic blocks contiguous and eliminate unnecessary branches. 3-5

Assembler/Compiler Coding Rule 2. (M impact, ML generality) Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch. 3-5

Assembler/Compiler Coding Rule 3. (M impact, H generality) Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target. 3-6

Assembler/Compiler Coding Rule 4. (MH impact, MH generality) Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns. 3-7

Assembler/Compiler Coding Rule 5. (MH impact, MH generality) Selectively inline a function if doing so decreases code size or if the function is small and the call site is frequently executed. 3-8

Assembler/Compiler Coding Rule 6. (ML impact, ML generality) If there are more than 16 nested calls and returns in rapid succession; consider transforming the program with inline to reduce the call depth. 3-8

Assembler/Compiler Coding Rule 7. (ML impact, ML generality) Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred. 3-8

Assembler/Compiler Coding Rule 8. (L impact, L generality) If the last statement in a function is a call to another function, consider converting the call to a jump. This will save the call/return overhead as well as an entry in the return stack buffer. 3-8

Assembler/Compiler Coding Rule 9. (M impact, L generality) Do not put more than four branches in a 16-byte chunk. 3-8

Assembler/Compiler Coding Rule 10. (M impact, L generality) Do not put more than two end loop branches in a 16-byte chunk. 3-8

Assembler/Compiler Coding Rule 11. (M impact, H generality) When executing code from the Decoded ICache, direct branches that are mostly taken should have all their instruction bytes in a 64B cache line and nearer the end of that cache line. Their targets should be at or near the beginning of a 64B cache line. 3-8

Assembler/Compiler Coding Rule 12. (M impact, H generality) If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, it should be placed on a different code page. 3-8

Assembler/Compiler Coding Rule 13. (M impact, L generality) When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path. 3-8

Assembler/Compiler Coding Rule 14. (H impact, M generality) Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop. 3-10

Assembler/Compiler Coding Rule 15. (M impact, M generality) Unroll loops that are frequently executed and have a predictable number of iterations to reduce the number of iterations to 16 or fewer. Do this unless it increases code size so that the working set no longer fits in the instruction cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches). 3-10

Assembler/Compiler Coding Rule 16. (ML impact, M generality) For improving fetch/decode throughput, Give preference to memory flavor of an instruction over the register-only flavor of the same instruction, if such instruction can benefit from micro-fusion. 3-11

USER RULES

Assembler/Compiler Coding Rule 17. (M impact, ML generality) Employ macrofusion where possible using instruction pairs that support macrofusion. Prefer TEST over CMP if possible. Use unsigned variables and unsigned jumps when possible. Try to logically verify that a variable is non-negative at the time of comparison. Avoid CMP or TEST of MEM-IMM flavor when possible. However, do not add other instructions to avoid using the MEM-IMM flavor. 3-15

Assembler/Compiler Coding Rule 18. (M impact, ML generality) Software can enable macro fusion when it can be logically determined that a variable is non-negative at the time of comparison; use TEST appropriately to enable macrofusion when comparing a variable with 0. 3-16

Assembler/Compiler Coding Rule 19. (MH impact, MH generality) Favor generating code using imm8 or imm32 values instead of imm16 values. 3-17

Assembler/Compiler Coding Rule 20. (M impact, ML generality) Ensure instructions using 0xF7 opcode byte does not start at offset 14 of a fetch line; and avoid using these instruction to operate on 16-bit data, upcast short data to 32 bits. 3-19

Assembler/Compiler Coding Rule 21. (MH impact, MH generality) Break up a loop body with a long sequence of instructions into loops of shorter instruction blocks of no more than the size of the LSD. 3-19

Assembler/Compiler Coding Rule 22. (M impact, M generality) Avoid putting explicit references to ESP in a sequence of stack operations (POP, PUSH, CALL, RET). 3-21

Assembler/Compiler Coding Rule 23. (ML impact, L generality) Use simple instructions that are less than eight bytes in length. 3-21

Assembler/Compiler Coding Rule 24. (M impact, MH generality) Avoid using prefixes to change the size of immediate and displacement. 3-21

Assembler/Compiler Coding Rule 25. (M impact, H generality) Favor single-micro-operation instructions. Also favor instruction with shorter latencies. 3-22

Assembler/Compiler Coding Rule 26. (M impact, L generality) Avoid prefixes, especially multiple non-0F-prefixed opcodes. 3-22

Assembler/Compiler Coding Rule 27. (M impact, L generality) Do not use many segment registers. 3-22

Assembler/Compiler Coding Rule 28. (M impact, M generality) Avoid using complex instructions (for example, enter, leave, or loop) that have more than four µops and require multiple cycles to decode. Use sequences of simple instructions instead. 3-22

Assembler/Compiler Coding Rule 29. (MH impact, M generality) Use push/pop to manage stack space and address adjustments between function calls/returns instead of enter/leave. Using enter instruction with non-zero immediates can experience significant delays in the pipeline in addition to misprediction. 3-22

Assembler/Compiler Coding Rule 30. (ML impact, L generality) If an LEA instruction using the scaled index is on the critical path, a sequence with ADDs may be better. 3-24

Assembler/Compiler Coding Rule 31. (ML impact, L generality) Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction. 3-26

Assembler/Compiler Coding Rule 32. (M impact, ML generality) Use dependency-breaking-idiom instructions to set a register to 0, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move 0 into the register instead. This requires more code space than using XOR and SUB, but avoids setting the condition codes. 3-27

Assembler/Compiler Coding Rule 33. (M impact, MH generality) Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using MOVZX. 3-28

Assembler/Compiler Coding Rule 34. (M impact, M generality) Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension. 3-28

Assembler/Compiler Coding Rule 35. (ML impact, L generality) Avoid placing instructions that use 32-bit immediates which cannot be encoded as sign-extended 16-bit immediates near each other. Try to schedule µops that have no immediate immediately before or after µops with 32-bit immediates. 3-29

USER RULES

Assembler/Compiler Coding Rule 36. (ML impact, M generality) Use the TEST instruction instead of AND when the result of the logical `and` is not used. This saves µops in execution. Use a `test` of a register with itself instead of a `cmp` of the register to zero, this saves the need to encode the zero and saves encoding space. Avoid comparing a constant to a memory operand. It is preferable to load the memory operand and compare the constant to a register. 3-29

Assembler/Compiler Coding Rule 37. (ML impact, M generality) Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a TEST instruction instead of a compare. Be certain that any code transformations made do not introduce problems with overflow. 3-29

Assembler/Compiler Coding Rule 38. (H impact, MH generality) For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies. 3-31

Assembler/Compiler Coding Rule 39. (M impact, ML generality) Avoid introducing dependences with partial floating-point register writes, e.g. from the MOVSD XMMREG1, XMMREG2 instruction. Use the MOVAPD XMMREG1, XMMREG2 instruction instead. 3-36

Assembler/Compiler Coding Rule 40. (H impact, M generality) Pass parameters in registers instead of on the stack where possible. Passing arguments on the stack requires a store followed by a reload. While this sequence is optimized in hardware by providing the value to the load directly from the memory order buffer without the need to access the data cache if permitted by store-forwarding restrictions, floating-point values incur a significant latency in forwarding. Passing floating-point arguments in (preferably XMM) registers should save this long latency operation. 3-48

Assembler/Compiler Coding Rule 41. (H impact, M generality) A load that forwards from a store must have the same address start point and therefore the same alignment as the store data. 3-48

Assembler/Compiler Coding Rule 42. (H impact, M generality) The data of a load which is forwarded from a store must be completely contained within the store data. 3-48

Assembler/Compiler Coding Rule 43. (H impact, ML generality) If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. This is better than incurring the penalties of a failed store-forward. 3-48

Assembler/Compiler Coding Rule 44. (MH impact, ML generality) Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed 3-48

Assembler/Compiler Coding Rule 45. (H impact, MH generality) Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction - for example, MUL or DIV. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain. 3-51

Assembler/Compiler Coding Rule 46. (M impact, MH generality) Calculate store addresses as early as possible to avoid having stores block loads. 3-52

Assembler/Compiler Coding Rule 47. (H impact, M generality) Try to arrange data structures so they permit sequential access. 3-54

Assembler/Compiler Coding Rule 48. (H impact, M generality) Make sure that the stack is aligned at the largest multi-byte granular data type boundary matching the register width. 3-54

Assembler/Compiler Coding Rule 49. (M impact, L generality) If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch. 3-55

Assembler/Compiler Coding Rule 50. (**H impact, L generality)** Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate 4-KByte pages or on separate aligned 1-KByte sub-pages. 3-55

USER RULES

Assembler/Compiler Coding Rule 51. (H impact, L generality) If an inner loop writes to more than four arrays (four distinct cache lines), apply loop fission to break up the body of the loop so only four arrays are being written to in each iteration of each of the resulting loops. 3-56

Assembler/Compiler Coding Rule 52. (H impact, M generality) Minimize changes to bits 8-12 of the floating-point control word. Changes for more than two values (each value being a combination of the following bits: precision, rounding and infinity control, and the rest of bits in FCW) leads to delays that are on the order of the pipeline depth. 3-72

Assembler/Compiler Coding Rule 53. (H impact, L generality) Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision, and infinity bits. 3-73

Assembler/Compiler Coding Rule 54. (H impact, L generality) Minimize the number of changes to the precision mode. 3-73

Assembler/Compiler Coding Rule 55. (M impact, M generality) Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack. 3-74

Assembler/Compiler Coding Rule 56. (H impact, M generality) Use the 32-bit versions of instructions in 64-bit mode to reduce code size unless the 64-bit version is necessary to access 64-bit data or additional registers. .. 13-1

Assembler/Compiler Coding Rule 57. (M impact, MH generality) When they are needed to reduce register pressure, use the 8 extra general purpose registers for integer code and 8 extra XMM registers for floating-point or SIMD code. 13-2

Assembler/Compiler Coding Rule 58. (ML impact, M generality) Prefer 64-bit by 64-bit integer multiplication that produces 64-bit results over multiplication that produces 128-bit results. 13-2

Assembler/Compiler Coding Rule 59. (ML impact, M generality) Stagger accessing the high 64-bit result of a 128-bit multiplication after accessing the low 64-bit results. 13-2

Assembler/Compiler Coding Rule 60. (ML impact, M generality) Use the 64-bit versions of multiply for 32-bit integer multiplies that require a 64 bit result. 13-6

Assembler/Compiler Coding Rule 61. (ML impact, M generality) Use the 64-bit versions of add for 64-bit adds. 13-6

Assembler/Compiler Coding Rule 62. (L impact, L generality) If software prefetch instructions are necessary, use the prefetch instructions provided by SSE. 13-6

Assembler/Compiler Coding Rule 63. (H impact, H generality) Whenever a 256-bit AVX code block and 128-bit SSE code block might execute in sequence, use the VZEROUPPER instruction to facilitate a transition to a “Clean” state for the next block to execute from. 15-11

Assembler/Compiler Coding Rule 64. (H impact, M generality) Align data to 32-byte boundary when possible. Prefer store alignment over load alignment. 15-24

Assembler/Compiler Coding Rule 65. (M impact, H generality) Align data to 32-byte boundary when possible. If it is not possible to align both loads and stores, then prefer store alignment over load alignment. 15-26

Assembler/Compiler Coding Rule 66. (M impact, M generality) Use Blend instructions in lieu of shuffle instruction in AVX whenever possible. 15-35

TUNING SUGGESTIONS

Tuning Suggestion 1. In rare cases, a performance problem may be caused by executing data on a code page as instructions. This is very likely to happen when execution is following an indirect branch that is not resident in the trace cache. If this is clearly causing a performance problem, try moving the data elsewhere, or inserting an illegal opcode or a `pause` instruction immediately after the indirect branch. Note that the latter two alternatives may degrade performance in some circumstances. 3-55

Tuning Suggestion 2. Optimize single threaded code to maximize execution throughput first. 11-28

Tuning Suggestion 3. Employ efficient threading model, leverage available tools (such as Intel Threading Building Block, Intel Thread Checker, Intel Thread Profiler) to achieve optimal processor scaling with respect to the number of physical processors or processor cores. 11-28

TUNING SUGGESTIONS