OpenCL™ Developer Guide for Intel® Processor Graphics

Contents

OpenCL™ Optimization Guide for Visual Computing Systems 1

Apply the optimizations described in this guide using the Intel® OpenCL™ Code Builder, which is available with:

1. **Intel® Media Server Studio**
2. **Intel® Code builder for OpenCL™ API for Linux***

NOTE This publication, the OpenCL™ Developer Guide for Intel® Processor Graphics, was previously known as the OpenCL™ Optimization Guide for Intel® Processor Graphics.

OpenCL™ Code builder is a software development tool that enables development of OpenCL applications via well-known integrated development environments, targeting the Intel® Architecture processors with the Intel® Processor Graphics.

The tool supports local (host-based) and remote (target-based) development on the following platforms, IDEs, and devices:

Operating System	Host/Target	OpenCL™ Code Builder as part of Intel® Media Server Studio	Intel® Code Builder for OpenCL™ API
Windows*	Host	-	Yes
	Target	-	Yes
Linux*	Host	Yes	Yes
	Target	Yes	Yes

To better understand which version of Code Builder is fitting to you, check the Which Version of the Code Builder to Pick? article.

Legal Information

Getting Help and Support

You can get support with issues you encounter through the Intel® Code Builder for OpenCL™ API support forum at intel.com/software/opencl/.

For information on SDK requirements, known issues and limitations, refer to the OpenCL™ Code Builder - Release Notes.

Introduction

- About this Document
- Basic Concepts
- Using Data Parallelism
- Related Products

About this Document

The OpenCL™ Code Builder - Optimization Guide describes the optimization guidelines of OpenCL applications targeting the Intel CPUs with Intel® Graphics. **If your application targets Intel® Xeon™ processors and Intel® Xeon Phi™ coprocessors, refer to the**OpenCL Optimization Guide for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors.

The SDK extends Intel support of open standards to include certified OpenCL 2.0 support for Intel Architecture processors on Microsoft Windows 7* and Windows 8* operating systems. Refer to See Also section for details on OpenCL 2.0 support. The implementation also enables utilizing the compute resources of both the Intel CPU and Intel Graphics simultaneously.

The guide provides tips for writing optimal OpenCL code and introduces the essential steps to identifying sections of code that consume the most compute cycles.

This document targets OpenCL developers and assumes you understand the basic concepts of the OpenCL standard.

For details on OpenCL 2.0 support on Intel Architecture CPU and Intel Graphics, refer to the SDK User Manual or Release Notes. To get started with important OpenCL 2.0 features including Shared Virtual Memory (SVM), refer to the links in the **See Also** section below.

See Also

OpenCL Code Builder - Release Notes

User Manual - OpenCL™ Code Builder

>Get Started with OpenCL 2.0 API

>

The OpenCL 2.0 Specification at http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

Overview Presentations of the OpenCL Standard at http://www.khronos.org/registry/

Basic Concepts

The following are the basic OpenCL™ concepts used in this document. The concepts are based on notions in OpenCL specification that defines:

- *Compute unit* - an OpenCL device has one or more compute units. A work-group executes on a single compute unit. A compute unit is composed of one or more processing elements and local memory. A compute unit can also include dedicated texture sampling units that can be accessed by its processing elements.
- *Device* - a collection of compute units.
- *Command-queue* is an object that holds commands to be executed on a specific device. Examples of commands include executing kernels, or reading and writing memory objects.
- *Kernel* - a function declared in an OpenCL program and executed on an OpenCL device. A kernel is identified by the __kernel or kernel qualifier applied to any function defined in a program.
- *Work-item* - one of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executed work-items within the collection by its global ID and local ID.
- *Work-group* - a collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers. Each work-group has the following properties:
- - Data sharing between work-items via local memory
  - Synchronization between work-items via barriers and memory fences
  - Special work-group level built-in functions, such as work_group_copy.

When launching the kernel for execution, the host code defines the grid dimensions, or the global work size. The host code can also define the partitioning to work-groups, or leave it to the implementation. During the execution, the implementation runs a single work-item for each point on the grid. It also groups the execution on compute units according to the work-group size.

The order of execution of work-items within a work-group, as well as the order of work-groups, is implementation-specific.

See Also

User Manual - OpenCL™ Code Builder

Overview Presentations of the OpenCL Standard at http://www.khronos.org/registry/

Using Data Parallelism

The OpenCL™ standard basic data parallelism uses the Single Program Multiple Data (SPMD) technique. SPMD resembles fragment processing with pixel shaders in the context of graphics.

In this programming model, a kernel executes concurrently on multiple elements. Each element has its own data and its own program counter. If elements are vectors of any kind (for example, four-way pixel values for an RGBA image), consider using vector types.

This section describes how to convert regular C code to an OpenCL program using a simple "hello world" example. Consider the following C function:

```
void scalar_mul(int n, const float *a, const float *b, float *result)
{
        int i;
        for (i = 0; i < n; ++i)
        result[i] = a[i] * b[i];
}
```

This function performs element-wise multiplication of two arrays, a and b. Each element in result stores the product of the corresponding elements from arrays a and b.

Consider the following:

- The for loop consists of two parts: the loop statement that defines the range of operation (a single dimension containing n elements), and the loop body itself.
- The basic operation is done on scalar variables (float data types).
- Loop iterations are independent.

The same function in OpenCL appears as follows:

```
__kernel void scalar_mul(__global const float *a,
        __global const float *b,
        __global float *result)
{
        int i = get_global_id(0);
        result[i] = a[i] * b[i];
}
```

The kernel function performs the same basic element-wise multiplication of two scalar variables. The index is provided by use of a built-in function that gives the global ID, a unique number for each work-item within the grid defined by the NDRange.

The code itself does not imply any parallelism. Only the combination of the code with the execution over a global grid implements the parallelism of the device.

This parallelization method abstracts the details of the underlying hardware. You can write your code according to the native data types of the algorithm. The implementation takes care of the mapping to specific hardware.

Related Products

The following is the list of products related to Intel® SDK for OpenCL™ Applications 2014.

- Intel Graphics Performance Analyzers (Intel GPA)
- Intel VTune™ Amplifier XE
- Intel Media SDK
- Intel Perceptual Computing SDK

Coding for the Intel® Processor Graphics

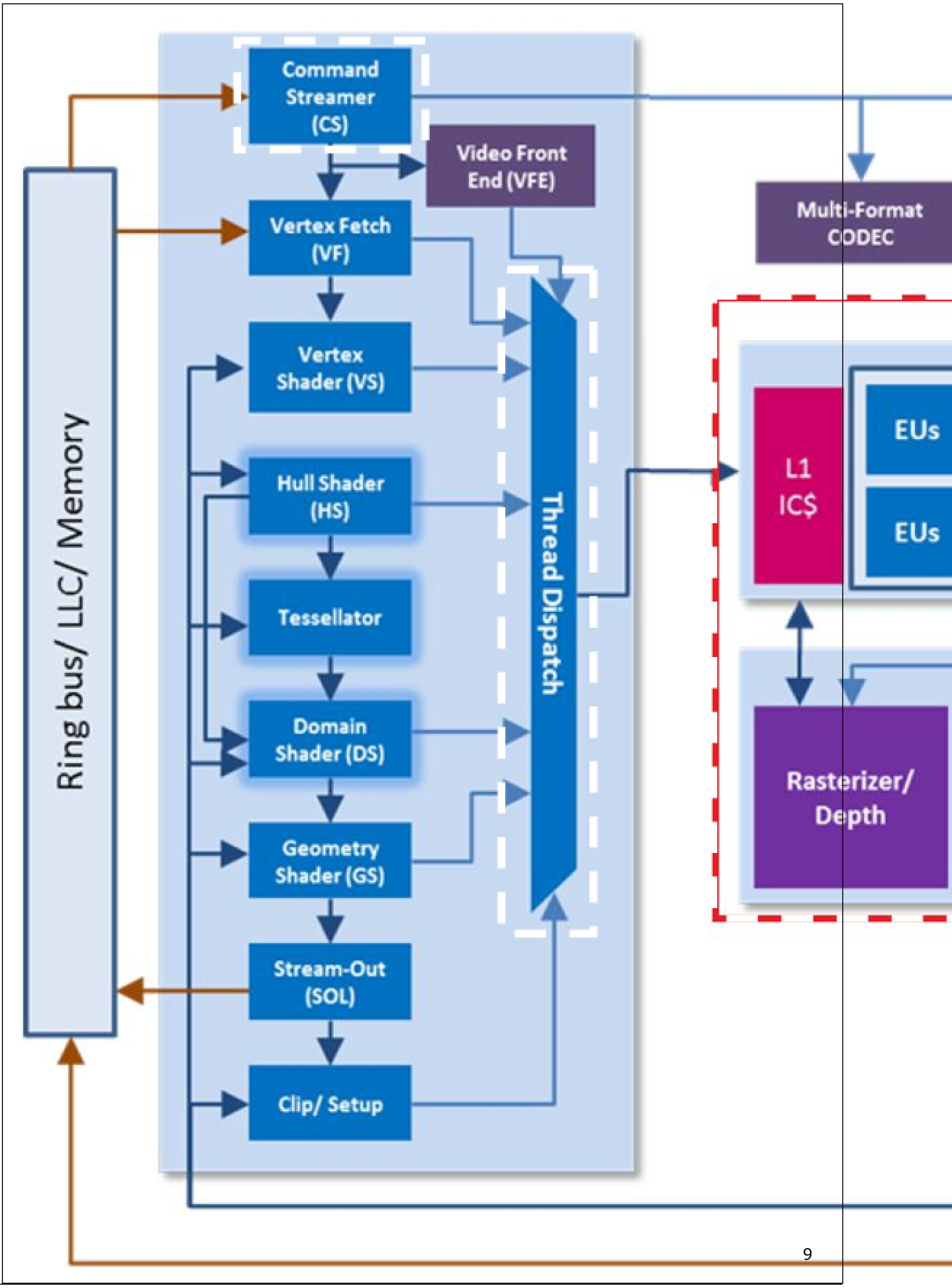
Execution of OpenCL™ Work-Items: the SIMD Machine

This chapter overviews the Compute Architecture of the Intel® Graphics and its component building blocks. For more details please refer to the references in the See Also section. The Intel Graphics device is equipped with several Execution Units (EUs), while each EU is a multi-threaded SIMD processor. Compiler generates SIMD code to map several work-items to be executed simultaneously within a given hardware thread. The SIMD-width for kernel is a heuristic driven compiler choice. SIMD-8, SIMD-16, SIMD-32 are common SIMD-width examples.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction, then the SIMD lanes can be maximally utilized. If one or more of the kernel instances choose a divergent branch, then the thread executes the two paths of the branch and merges the results by mask. The EUs branch unit keeps track of such branch divergence and branch nesting.

Command Streamer and Global Thread Dispatcher logic are responsible for thread scheduling; see the part, highlighted with the white dashed line of the Figure 1.

Figure 1



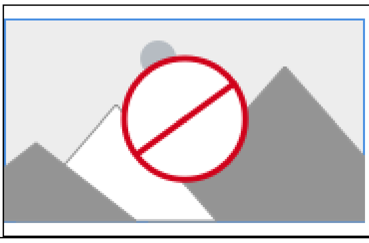
An example product based on Intel Graphics Compute Architecture. To simplify the picture, the low-end instantiation composed of one slice (with just one subslice), in red dashed rectangle, is shown. Together, execution units, subslices, and slices are the modular building blocks that are composed to create many product variants.

The building block of the architecture is the **execution unit**, commonly abbreviated as just EU. EUs are Simultaneous Multi-Threading (**SMT**) compute processors that drive multiple issuing of the Single Instruction Multiple Data Arithmetic Logic Units (**SIMD**). The highly threaded nature of the EUs ensures continuous streams of ready-to-execute instructions, while also enabling latency hiding of longer operations such as memory requests.

A group of EUs constitute a "sub-slice". The EUs in a sub-slice share:

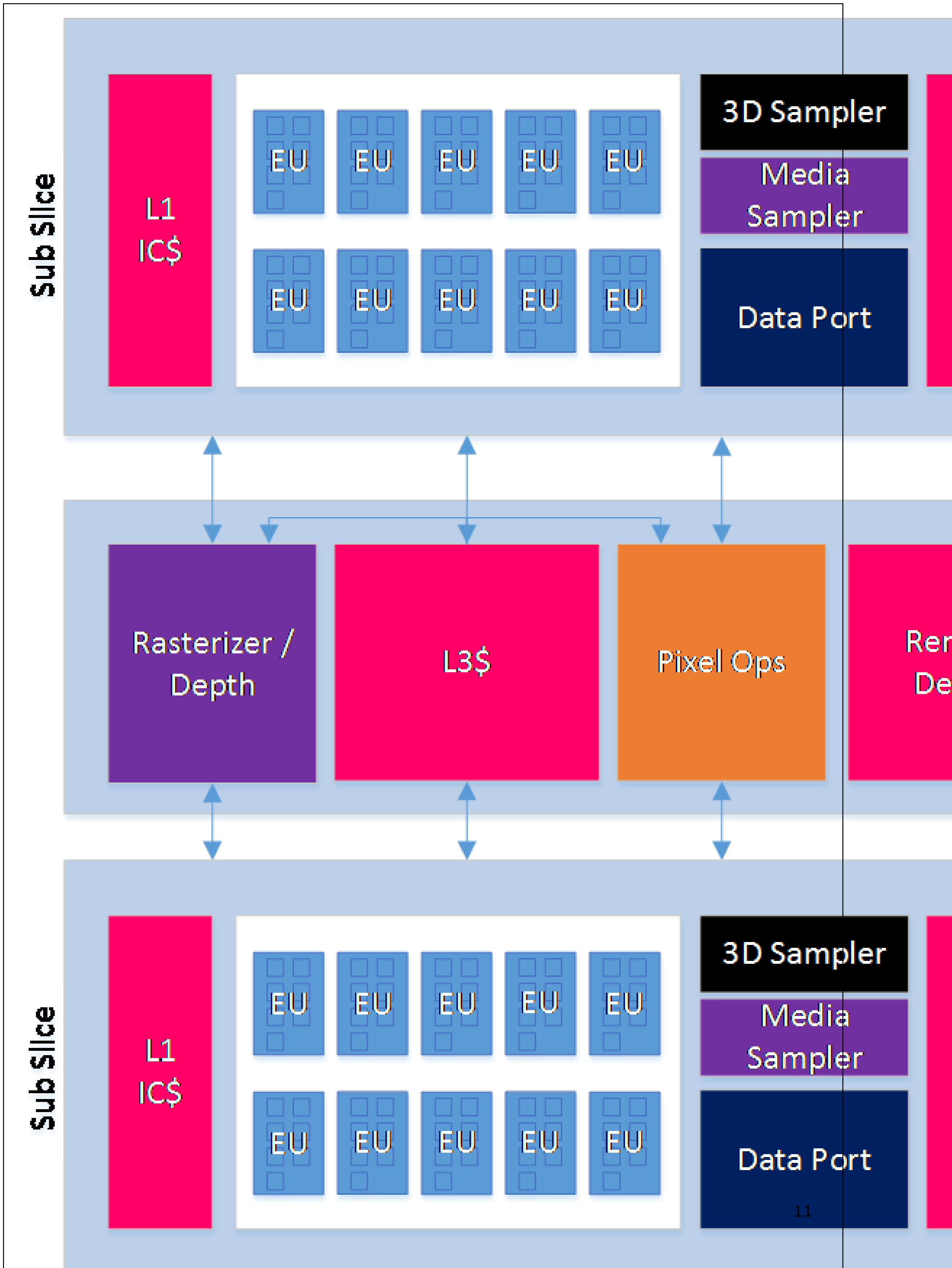
- Texture sampler and L1 and L2 texture caches, which are the path for accessing OpenCL images
- Data port (general memory interface), which is the path for OpenCL buffers
- Other hardware blocks like instruction cache

Figure 2. Subslice, a cluster of Execution Units, instantiating common Sampler and Data Port units.



In turn, one sub-slice (see red-dashed part of the *Figure 1*) in the low-end GPUs or more sub-slices (see *Figure 3*) for a more regular case, constitute the slice that adds L3 cache (for OpenCL buffers), Shared Local Memory (SLM), and Barriers as common assets.

Figure 3. The slice of Intel® Graphics, containing two Subslices. The Slice adds L3 cache, shared local memory, atomics, barriers, and other supporting fixed function.



The number of (sub-) slices and EUs, numbers of samplers, total amount of SLM, and so on depends on SKU and generation of the Intel® Graphics device. You can query these values with the regular clGetDeviceInfo routine, for example, with CL_DEVICE_MAX_COMPUTE_UNITS or other parameters. For details on memory and caches for the Intel Graphics, refer to the "Memory Access Considerations" section.

Given the high number of EUs, multi-threading and SIMD within an EU, is it important to follow the work-group recommendations in order to fully saturate the device. See the "Work-Group Size Recommendations Summary" section for the details.

For further details on the architecture, please refer to the Compute Architecture of Intel Processor Graphics Gen7.5 and Gen8 whitepapers referenced in the See Also section.

See Also

More on the Gen7.5 and Gen8 Compute Architectures: https://software.intel.com/en-us/articles/intel-graphics-developers-guides

Work-Group Size Recommendations Summary

Introduction to OpenCL Code Builder and deep dive to Intel Iris Graphics compute architecture

Memory Hierarchy

Intel® Graphics Compute Architecture uses system memory as a compute device memory. Such memory is unified by means of sharing the same DRAM with the CPU. The obvious performance advantage is that shared physical memory enables zero-copy transfers between host CPU and Intel® Graphics OpenCL™ device. The same zero-copy path works for the CPU OpenCL™ device and finally for the CPU-GPU shared context. Refer to the "Mapping Memory Objects" section for more information.

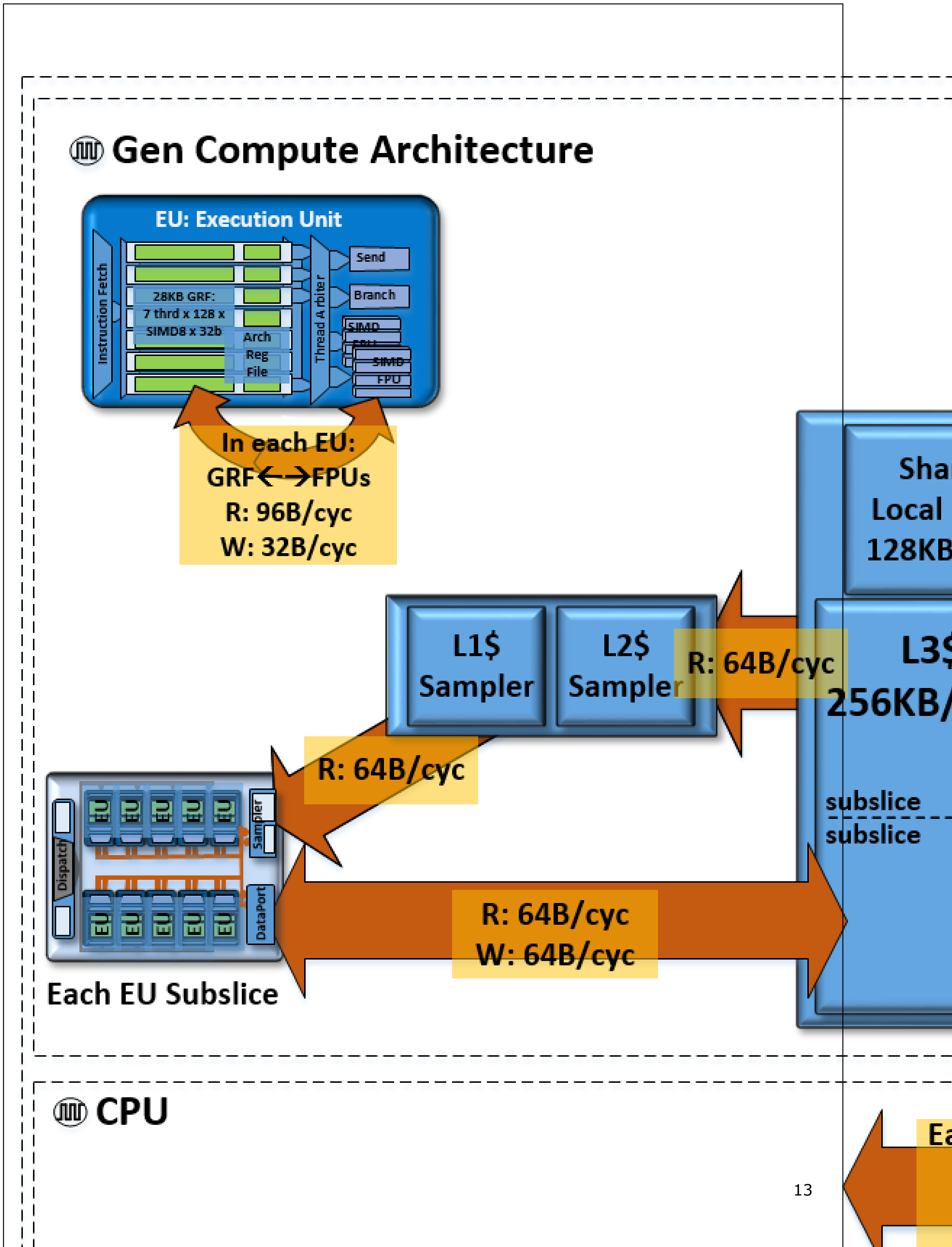
The Compute Architecture memory system is augmented with several levels of caches:

- Read-only memory path for OpenCL images which includes a level-1 (**L1**) and a level-2 (**L2**) sampler caches. Image writes follow different path (see below);
- Level-3 (**L3**) data cache is a slice-shared asset. All read and write actions on OpenCL buffers flows through the L3 data cache in units of 64-byte wide cache lines. The L3 cache includes sampler read transactions that are missing in the L1 and L2 sampler caches, and also supports sampler writes. See section "Execution of OpenCL™ Work-Items: the SIMD Machine" for details on slice-shared assets.
- **NOTE** The L3 efficiency is highest for accesses that are cache line-aligned and adjacent within cache line
- **Shared Local Memory (SLM)** is a dedicated structure within the L3 that supports the work-group local memory address space. The read/write bus interface to shared local memory is again 64-bytes-wide. But shared local memory is organized as 16 banks at 4-byte granularity. This organization can yield full bandwidth access for access patterns that may not be 64-byte aligned or that may not be contiguously adjacent in memory.
- **NOTE** The amount of SLM is an important limiting factor for the number of work-groups that can be executed simultaneously on the device. Use the clGetDeviceInfo(…CL_DEVICE_LOCAL_MEM_SIZE) call to query the exact value.

NOTE As shared local memory is highly banked, it is more important to minimize bank conflicts when accessing local memory than to minimize the number of cache lines.

Finally, the entire architecture interfaces to the rest of the SoC components via a dedicated interface unit called the Graphics Technology Interface (GTI). The rest of SoC memory hierarchy includes the large Last-Level Cache (LLC, which is shared between CPU and GPU), possibly embedded DRAM and finally the system DRAM.

Figure 4. View of memory hierarchy and peak bandwidths (in bytes/cycle) for the Gen7.5 compute architecture (4th Generation Intel® Core™ family of microprocessors).



Please find more details on the memory access in the following sections.

See Also

Mapping Memory Objects

Memory Access Overview

Global Memory Size

More on the Gen7.5 and Gen8 Compute Architectures: https://software.intel.com/en-us/articles/intel-graphics-developers-guides

Introduction to OpenCL Code Builder and deep dive to Intel Iris Graphics compute architecture

Platform-Level Considerations

- Intel® Turbo Boost Technology Support
- Global Memory Size

Intel® Turbo Boost Technology Support

Intel® Turbo Boost Technology applies to CPU cores and to the Intel Graphics device. The Intel Graphics and the CPU must run within the thermal constraints of the system. Thus, either the CPU or the Intel Graphics might boost or throttle the frequency as needed.

Refer to the Intel® Turbo Boost Technology website for the list of Intel processors that support the technology.

NOTE Frequency change on one device impacts the other. Too intensive application polling (such as looping on some flag waiting for the Intel Graphics to complete the job) can cause an increase in CPU frequency, which can negatively impact the Intel Graphics performance.

Global Memory Size

OpenCL™ “global” memory is allocated from system (host) memory for the CPU and the Intel® Graphics devices. The amount of available memory depends on the amount of computer system memory and the operating system (32 or 64 bit). For example, a system with 4GB of RAM running on a 32-bit OS usually has less than 3GB available for system memory. This impacts the amount of global memory available for the Intel® Processor Graphics and CPU device. Use the clGetDeviceInfo(…,CL_DEVICE_GLOBAL_MEM_SIZE) query to get information on the *total* available amount of memory. Notice that the maximum size of an individual memory allocation for the device can be queried with clGetDeviceInfo(…,CL_DEVICE_MAX_MEM_ALLOC_SIZE).

Your code should handle the failures to allocate resources, for example manifested by CL_OUT_OF_RESOURCES error.

Global memory performance depends on the frequency of DDR memory.

Since global memory is shared between the CPU and the Intel® Processor Graphics, it is important to use mapping for memory objects (see the "Mapping Memory Objects" section).

See Also

Mapping Memory Objects

Application-Level Optimizations

Minimizing Data Copying

The application should process data “in-place” and minimize copying memory objects. For example, OpenCL™ 1.2 and lower requires the global work dimensions be exact multiples of the local work-group dimensions. For a typical image processing task, require the work-groups to be tiles that exactly cover a frame buffer. If the global size differs from the original image, you might decide to copy and pad the original image buffer, so the kernel does not need to check every work-item to see if it falls outside the image. But this can add several milliseconds of processing time just to create and copy images. Refer to the section "Avoid Handling Edge Conditions in Kernels" for alternatives, including most elegant solution with OpenCL 2.0.

See Also

Avoiding Handling Edge Conditions in Kernels

Avoiding Needless Synchronization

For best results, try to avoid explicit command synchronization primitives (such as clEnqueueMarker or Barrier), also explicit synchronization commands and event tracking result in cross-module round trips, which decrease performance. The less you use explicit synchronization commands, the better the performance.

Use the following techniques to reduce explicit synchronization:

- Continue executing kernels until you really need to read the results; this idiom best expressed with in-order queue and blocking call to clEnqueueMapXXX or clEnqueueReadXXX.
- If an in-order queue expresses the dependency chain correctly, exploit the in-order queue rather than defining an event-driven string of dependent kernels. In the in-order execution model, the commands in a queue are automatically executed back-to-back, in the order of submission. This suits very well a typical case of a processing pipeline. Consider the following recommendations:
  - Avoid any host intervention to the in-order queue (like blocking calls) and additional synchronization costs.
  - When you have to use the blocking API, use OpenCL™ API, which is more effective than explicit synchronization schemes, based on OS synchronization primitives.
  - If you are optimizing the kernel pipeline, first measure kernels separately to find the most time-consuming one. Avoid calling clFinish or clWaitForEvents frequently (for example, after each kernel invocation) in the final pipeline version. Submit the whole sequence (to the in-order queue) and issue clFinish (or wait on the event) once. This reduces host-device round trips.
  - Consider OpenCL 2.0 “enqueue_kernel” feature that allows a kernel to independently enqueue to the same device, without host interaction. Notice that this approach is useful not just for recursive kernels, but also for regular non-recursive chains of the lightweight kernels. Refer to the **See Also** section below.

See Also

GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions

Reusing Compilation Results with clCreateProgramWithBinary

If compilation time for an OpenCL™ program is of concern, consider reusing compilation results. It is typically faster than recreating you program from the source, but you should check if this is true for your specific program and device.

To retrieve binaries generated from calls to clCreateProgramWithSource and clBuildProgram, you can call clGetProgramInfo with the CL_PROGRAM_BINARIES parameter. For the performance-critical applications that typically precompile kernel code to an intermediate representation (IR), you can cache the resulting binaries after the first OpenCL™ compilation and reuse them on subsequent executions by calling clCreateProgramWithBinary. Another way to save intermediate binaries is to use the OpenCL Code builder, as described in the *User Manual - OpenCL™ Code Builder*.

NOTE Intermediate representations are different for the CPU and the Intel® Graphics devices.

See Also

User Manual - OpenCL™ Code Builder

Interoperability with Other APIs

- Interoperability between OpenCL and OpenGL
- Using Microsoft DirectX* Resources
- Aligning Pointers to Microsoft DirectX* Buffers Upon Mapping
- Note on Working with other APIs
- Note on Intel® Quick Sync Video

Interoperability between OpenCL and OpenGL

It is important to follow the right approach for OpenCL™-OpenGL* interoperability, taking into account limitations such as texture usages and formats and caveats like synchronization between the APIs. Also, the approach to interoperability (direct sharing, PBO-based, or plain mapping) might be different depending on the target OpenCL device. For Intel® HD Graphics and Intel Iris™ Pro Graphics OpenCL devices, the direct sharing referenced below is ultimately the right way to go.

See Also

OpenCL™ and OpenGL* Interoperability Tutorial at https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial

Using Microsoft DirectX* Resources

When you create certain types of the Microsoft DirectX* 10 or 11 resources, intended for sharing with Intel® Graphics OpenCL™ device using the cl_khr_d3d10 or cl_khr_d3d11 extension, you need to set the D3D10_RESOURCE_MISC_SHARED or D3D11_RESOURCE_MISC_SHARED flag. Use this flag for 2D non-mipmapped textures and do not use for other types of resources, like buffers.

See Also

Align Pointers to the Microsoft Direct X10 Buffers Upon Mapping

Aligning Pointers to Microsoft DirectX* Buffers Upon Mapping

If your application utilizes resource sharing with Microsoft DirectX* 10 or 11 by use of mapping, use the CL_MEM_USE_HOST_PTR flag. Also note that mapping is less efficient than using the cl_khr_d3d10_sharing or cl_khr_d3d11_sharing.

Consider the general interoperability flow:

1. Map a resource to CPU
2. Create a buffer wrapping the memory
3. Use the buffer by use of the OpenCL™ regular commands

4. Upon competition of the OpenCL commands, the resource can be safely unmapped.

Now use the CL_MEM_USE_HOST_PTR flag to directly operate on the mapped memory and avoid data copying upon OpenCL buffer creation. This method requires properly aligned memory. See the “Mapping Memory Objects” section for more information.

NOTE Aligning might result in unprocessed data between original and aligned pointer.

If it is acceptable for your application, and/or the copying overhead is of concern, consider aligning of the pointer returned by the DirectX map call to comply with the "Mapping Memory Objects" section.

Another potential workaround is to allocate larger DirectX resource than it is required, so that you have some room for a safe alignment. This approach requires some additional application logic.

See Also

Mapping Memory Objects

Using Microsoft DirectX* Resources

Note on Working with other APIs

Interoperability with the APIs like Microsoft DirectX* or Intel® Media SDK are managed through extensions. Extensions are associated with specific devices. For more information on the extensions, status of extension support on CPU and Intel® Graphics devices, and shared context, refer to the *OpenCL™ Code Builder - User’s Guide*.

Intel SDK for OpenCL Application samples demonstrate various interoperability options. You can download samples from the product page at intel.com/software/opencl/.

Measure the overheads associated with various acquiring or releasing of DirectX, OpenGL*, Intel Media SDK APIs and other resources. High costs like several milliseconds for a regular HD frame might indicate some implicit copying.

See Also

Using Microsoft DirectX* Resources

User Manual - OpenCL™ Code Builder

Note on Intel® Quick Sync Video

Check and adjust the device load when dealing with transcoding pipelines. For example running the Intel® Quick Sync Video encoding might reduce benefits of using the Intel® Graphics for some OpenCL™ frame preprocessing. The reason is that the Intel® Quick Sync Video encoding already loads the Intel® Graphics units quite substantially. In some cases using the CPU device for OpenCL tasks reduces the burden and improves the overall performance. Consider experimenting to find the best solution.

See Also

OpenCL™ and Intel® Media SDK Interoperability sample

Optimizing OpenCL™ Usage with Intel® Processor Graphics

- Optimizing Utilization of Execution Units
- Work-Group Size Recommendations Summary
- Memory Access Considerations
- Using Loops

Optimizing Utilization of Execution Units

When you tune your programs for execution on the Intel® Graphics device to improve performance, be aware of the way your kernels are executed on the hardware:

- Optimize the number of work-groups
- Optimize the work-group size
- Use barriers in kernels wisely
- Optimize thread utilization

The primary goal of every throughput computing machine is to keep a sufficient number of work-groups active, so that if one is stalled, another can run on its hardware resource.

The primary things to consider:

- Launch enough work items to keep EU threads busy, keep in mind that compiler may pack up to 32 work items per thread (with SIMD-32).
- In short/lightweight kernels: use short vector data types and compute multiple pixels to better amortize thread launch cost.

Work-Group Size Recommendations Summary

If your kernel uses local memory and/or barriers, the actual number of work-groups that can run simultaneously on one of the Intel® Graphics sub-slice is limited by the following key factors:

- There are 16 barrier registers per sub-slice, so no more than 16 work-groups can be executed simultaneously.
- The amount of shared local memory available per sub-slice (64KB). If for example a work-group requires 32KB of shared local memory, only 2 of those work-groups can run concurrently, regardless of work-group size.

Therefore, to keep the device utilization high with the limited number of workgroups, larger workgroup sizes are required. Use power-of-two workgroup sizes between 64 and 256.

The number of sub-slices depends on the hardware generation and specific product. Refer to the See Also section for the details of the architecture.

NOTE A bare minimum SLM allocation size is 4k per workgroup, so even if your kernel requires less bytes per work-group, the actual allocation still will be 4k. To accommodate many potential execution scenarios try to minimize local memory usage to fit the optimal value of 4K per workgroup. Also notice that the granularity of SLM allocation is 1K.

If your kernel is not using local memory or barriers, these restrictions do not apply, and work-group size of 32 work-items is optimal for the most cases.

NOTE Try different local sizes to find the value that provides better performance. You can leave the “local group size” to clEnqueueNDRangeKernel() specified as NULL, enabling the system to choose the work-group size.

See Also

More on the Gen7.5 and Gen8 Compute Architectures: https://software.intel.com/en-us/articles/intel-graphics-developers-guides

Memory Access Considerations

Memory Access Overview

Optimizing memory accesses is the first step to achieving high performance with OpenCL™ on the Intel® Graphics. Tune your kernel to access memory at an optimal granularity and with optimal addresses.

The OpenCL™ implementation for the Intel® Graphics primarily accesses global and constant memory through the following caches:

- GPU-specific L3 cache
- CPU and GPU shared Last Level Cache (LLC).

Of these two caches, it is important to optimize memory accesses for the L3 cache. L3 cache line is 64 bytes.

Finally, there are L1 and L2 caches that are specific to the sampler and renderer.

Accesses to __global memory and __constant memory go through the L3 cache and LLC. In addition, __private memory that spill from registers do the same. If multiple OpenCL work-items in the same hardware thread make requests to the same L3 cache line, these requests are collapsed to a single request. This means that the effective __global memory, __constant memory, and __private memory bandwidth is determined by the number of the accessed L3 cache lines that are accessed.

For example, if two L3 cache lines are accessed from different work items in the same hardware thread, memory bandwidth is one half of the memory bandwidth in case when only one L3 cache line is accessed.

__local memory is allocated directly from the L3 cache, and is divided into 16 banks at a 32-bit granularity. Because it is so highly banked, it is more important to minimize bank conflicts when accessing local memory than to minimize the number of L3 cache lines accesses.

All memory can be accessed in 8-bit, 16-bit, or 32-bit quantities. 32-bit quantities can be accessed as vectors of one, two, three, or four components.

Recommendations

Granularity

For all memory address spaces, to optimize performance, a kernel must access data in at least 32-bit quantities, from addresses that are aligned to 32-bit boundaries. A 32-bit quantity can consist of any type, for example:

- char4s
- ushort2s
- ints

These data types can be accessed with identical memory performance. If possible, access up to four 32-bit quantities (float4, int4, etc) at a time to improve performance. Accessing more than four 32-bit quantities at a time may reduce performance.

__global Memory and __constant Memory

To optimize performance when accessing __global memory and __constant memory, a kernel must minimize the number of cache lines that are accessed.

However, if many work-items access the same global memory or constant memory array element, memory performance may be reduced.

For this reason, move frequently accessed global or constant data, such as look-up tables or filter coefficients, to local or private memory to improve performance.

If a kernel indexes memory, where index is a function of a work-item global id(s), the following factors have big impact on performance:

- The work-group dimensions
- The function of the work-item global id(s).

To see how the work-group dimensions can affect memory bandwidth, consider the following code segment:

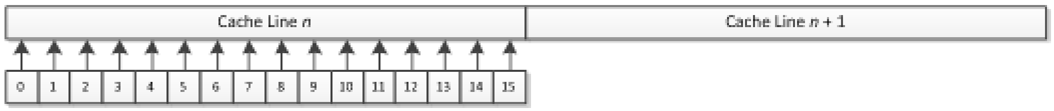
```
__global int*   myArray = ...;
uint            myIndex = get_global_id(0) + get_global_id(1) * width;
int i = myArray [ myIndex ];
```

This is a typical memory access pattern for a two-dimensional array.

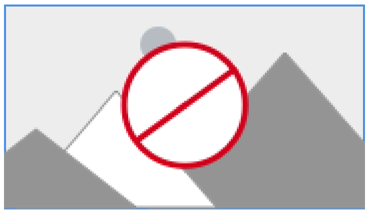
Consider three possible work-group dimensions, each describing a work-group of sixteen work-items:

- A "row" work-group: <16, 1, 1>
- A "square" work-group: <4, 4, 1>
- A "column" work-group: <1, 16, 1>

With the "row" work-group, get_global_id(1) is constant for all work-items in the work-group. myIndex increases monotonically across the entire work-group, which means that the read from myArray comes from a single L3 cache line (16 x sizeof(int) = 64 bytes).



With the "square" work-group, get_global_id(1) is different for every four work-items in the work-group. Within each group of four work-items, myIndex is monotonically increasing; the read from myArray comes from a different L3 cache line for each group of four work-items. Since four cache lines are accessed with the "square" work-group, this work-group sees 1/4th of the memory performance of the "row" work-group.

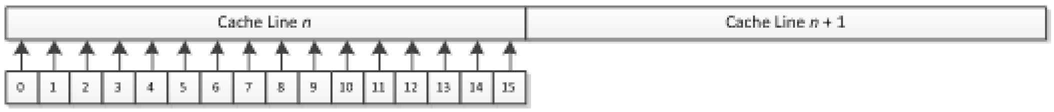


With the "column" work-group, get_global_id(1) is different for every work-item in the work-group; every read from myArray comes from a different cache line for every work-item in the work-group. If this is the case, 16 cache lines are accessed, and the column work-group sees 1/16th of the memory performance of the "row" work-group.

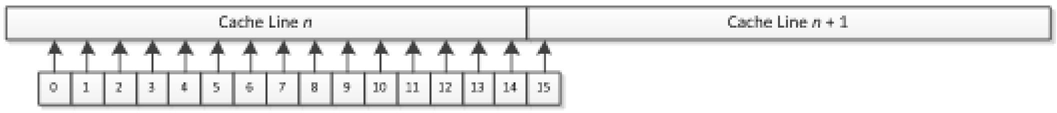
To see how the function of the work-item global ids can affect memory bandwidth, consider the following examples (assume a "row" work-group, < 16, 1, 1 >):

```
__global int*   myArray = ...;
int x;
x = myArray[ get_global_id(0) ];                                // case 1
x = myArray[ get_global_id(0) + 1 ];                    // case 2
x = myArray[ get_global_size(0) - 1 - get_global_id(0) ];       // case 3
x = myArray[ get_global_id(0) * 4 ];                    // case 4
x = myArray[ get_global_id(0) * 16 ];                   // case 5
x = myArray[ get_global_id(0) * 32 ];                   // case 6
```

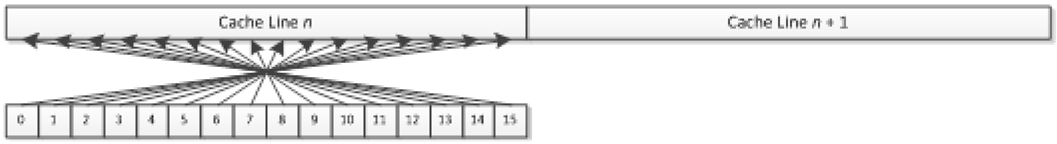
In Case 1, the read is cache-aligned, and the entire read comes from one cache line. This case should achieve full memory bandwidth.



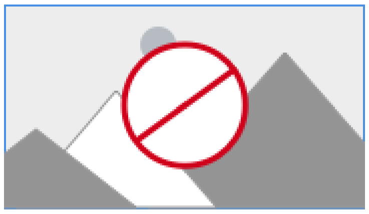
In Case 2, the read is not cache-aligned, so this read requires two cache lines, and achieves half of the memory performance of Case 1.



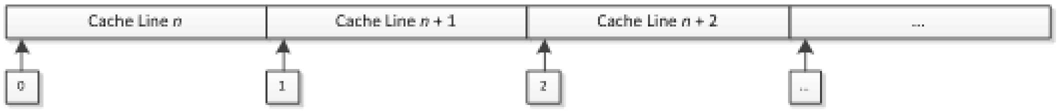
In Case 3, the addresses are decreasing instead of increasing, and they all come from the same cache line. This case achieves same memory performance as Case 1.



In Case 4, the addresses are stridden, so every fourth work-item accesses a new cache line. This case should achieve 1/4th of the memory performance of Case 1.



In both Case 5 and Case 6, each work-item is accessing a new cache line. Both of these cases provide similar performance, and achieve 1/16th of the memory performance of Case 1.



__private Memory

__private memory that is allocated to registers is typically very efficient to access. If the private memory doesn't fit in registers, however, the performance can be very poor. Since each work-item has its own spill space for __private memory, there is no locality for __private memory accesses, and each work-item frequently accesses a unique cache line for every access to __private memory. For this reason, accesses to __private memory data that has not been allocated to registers are very slow. In most cases, the compiler can map statically-indexed private arrays into registers. Also, in some cases, it can map dynamically-indexed private arrays in registers, but the performance of this code will be slightly lower than accessing statically indexed private arrays. As such, a common optimization is to modify code to ensure private arrays are statically indexed.

__local Memory

Local memory can be used to avoid multiple redundant reads from and writes to global memory. But it is important to note that the SLM (which is used to implement local memory), occupies the same place in the architecture as the L3 cache. So the performance of local memory accesses is often similar to that of a cache hit. Using local memory is typically only advantageous when the access pattern favors the banked nature of the SLM array.

When local memory is used to store temporary inputs and/or outputs, there are a few things to consider:

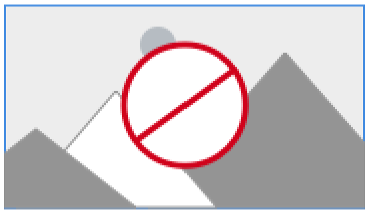
- When reading multiple items repeatedly from global memory:
- - You can benefit from prefetching global memory blocks into local memory once, incurring a local memory fence, and reading repeatedly from local memory instead.
  - Do not use single work-item (like the one with local id of 0) to load many global data items into the local memory by using a loop. Looped memory accesses are slow, and some items might be prefetched more than once.
  - Instead, designate work-items to prefetch a single global memory item each, and then incur a local memory fence, so that the local memory is full.
- When using local memory to reduce memory writes:
- - Enable a single work-item to write to an independent area of local memory space, and do not enable overlapping write operations.
  - If, for example, each work-item is writing to a row of pixels, the local memory size equals the number of local memory items times the size of a row, and each work-item indexes into its respective local memory buffer.

As we discussed earlier to optimize performance when accessing __local memory, a kernel must minimize the number of bank conflicts. As long as each work-item accesses __local memory with an address in a unique bank, the access occurs at full bandwidth. Work-items can read from the same address within a bank with no penalty, but writing to different addresses within the same bank produces a bank conflict and impacts performance.

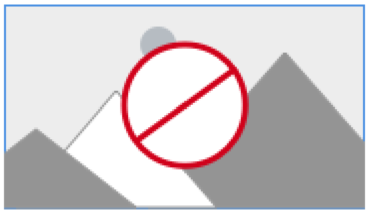
To see how bank conflicts can occur, consider the following examples (assume a "row" work-group, <16, 1, 1>):

```
__local int*    myArray = ...;
int x;
x = myArray[ get_global_id(0) ];                                 // case 1
x = myArray[ get_global_id(0) + 1 ];                     // case 2
x = myArray[ get_global_size(0) – 1 – get_global_id(0) ];        // case 3
x = myArray[ get_global_id(0) & ~1 ];                        // case 4
x = myArray[ get_global_id(0) * 2 ];                     // case 5
x = myArray[ get_global_id(0) * 16 ];                    // case 6
x = myArray[ get_global_id(0) * 17 ];                    // case 7
```

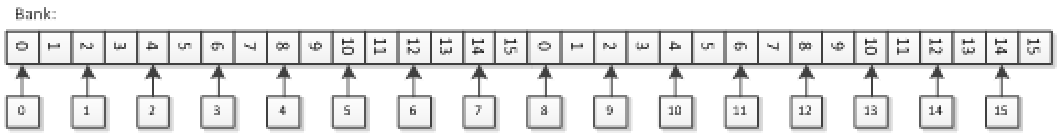
Cases 1, 2, and 3 access sixteen unique banks and therefore achieve full memory bandwidth. If you use global memory array instead of a local memory array, case 2 does not achieve full bandwidth due to accesses to two cache lines. The diagram below shows case 2.



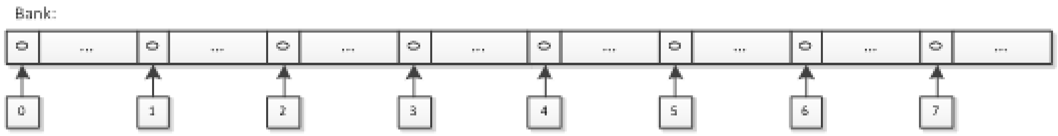
Case 4 reads from 8 unique banks, but with the same address for each bank, so it should also achieve full bandwidth.



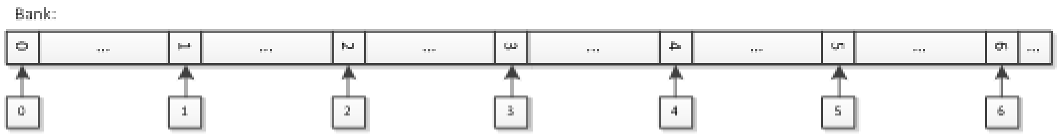
Case 5 reads from eight unique banks with a different address for each work-item, and therefore should achieve half of the bandwidth of Case 1.



Case 6 represents a worst-case for local memory: it reads from a single bank with a different address for each work-item. It should operate at 1/16th the memory performance of Case 1.



Case 7 is a stridden case similar to Case 6, but since it reads from 16 unique banks, this case also achieves full bandwidth.



The difference between Case 6 and Case 7 is important because this pattern is frequently used to access "columns" of data from a two-dimensional local memory array. Choose an array stride that avoids bank conflicts when accessing two-dimensional data from a local memory array, even if it results in a "wasted" column of data. For example, Case 7 has stride of 17 elements in compare to 16 elements in Case 6.

Kernel Memory Access Optimization Summary

A kernel should access at least 32-bits of data at a time, from addresses that are aligned to 32-bit boundaries. A char4, short2, int, or float counts as 32-bits of data. If you can, load two, three, or four 32-bit quantities at a time, which may improve performance. Loading more than four 32-bit quantities at a time may reduce performance.

Optimize __global memory and __constant memory accesses to minimize the number of cache lines read from the L3 cache. This typically involves carefully choosing your work-group dimensions, and how your array indices are computed from the work-item local or global id.

If you cannot access __global memory or __constant memory in an optimal manner, consider moving part of your data to __local memory, where more access patterns can execute with full performance.

Local memory is most beneficial when the access pattern favors the banked nature of the SLM hardware.

Optimize __local memory accesses to minimize the number of bank conflicts. Reading the same address from the same bank is OK, but reading different addresses from the same bank results in a bank conflict. Writes to the same bank always result in a bank conflict, even if the writes are going to the same address. Consider adding a column to two-dimensional local memory arrays if it avoids bank conflicts when accessing columns of data.

Avoid dynamically-indexed __private arrays if possible.

Using Loops

The Intel® Graphics device is optimized for code, which does not branch or loop. In the case, when a loop in a kernel is unavoidable, minimize the overhead by unrolling the loop either partially or completely in code, or using macros, and also minimize memory accesses within the loop.

The following example demonstrates partial unrolling of a loop in the example OpenCL™ kernel. Suppose you evaluate a polynomial, and you know that the order of the polynomial is a multiple of 4. Consider the following example:

```
__kernel void
poly(float *in, float *coeffs, float* result, int numcoeffs)
{
        // Un-optimized version
        int gid = get_global_id(0);
        result[gid] = 0;
        for(uint i=0; i<numcoeffs; i++) //numcoeffs is multiple of 4
        {
                result[gid] += pow(in[gid],i)*coeffs[i];
        }
}
```

The above code is an indeterminate loop—that is, the compiler does not know how many iterations the for loop executes. Furthermore, there are 3 memory accesses within each iteration of the loop, and the loop code must be executed each iteration. You can remove these overheads using partial loop unrolling and private variables, for example:

```
__kernel void
poly(float *in, float *coeffs, float* result, int numcoeffs)
{
        // Optimized version #1
        int gid = get_global_id(0);
        float result_pvt;
        float in_pvt = in[gid];
        result_pvt = 0;
        for(uint i=0; i<numcoeffs; i+=4) //numcoeffs is multiple of 4
        {
                result_pvt += pow(in_pvt,i)*coeffs[i];
                result_pvt += pow(in_pvt,i+1)*coeffs[i+1];
                result_pvt += pow(in_pvt,i+2)*coeffs[i+2];
                result_pvt += pow(in_pvt,i+3)*coeffs[i+3];
        }
        result[gid] = result_pvt;
}
```

In this optimized version, we divide the number of iterations by 4, and do only one memory access per original iteration. In any case where memory accesses can be replaced by private variables, this provides significant performance benefit. Furthermore, if multiple similar memory accesses are occurring in different kernels, then using shared local memory might provide performance gain. See section "Kernel Memory Access Optimization Summary" for details.

Another way to promote loop unrolling is to use macros to set constant loop iterations. The modified code:

```
__kernel void
poly(float *in, float *coeffs, float* result, int numcoeffs)
{
        // Optimized version #1
        int gid = get_global_id(0);
        float result_pvt;
        float in_pvt = in[gid];
        result_pvt = 0;
        for(uint i=0; i<NUMCOEFFS; i++)
        {
                result_pvt += pow(in_pvt,i)*coeffs[i];
        }
        result[gid] = result_pvt;
}
```

And from the host code, when compiling, use the flag:

```
-DNUMCOEFFS=16 // where 16 is the number of coefficients
```

It is possible when the loop iterations are known in advance, but you can also use this optimization to define the number of partial unrolls to use, in the case when you know a common denominator for all loop iterations.

When within a loop, use uint data types for iterations, as the Intel® Graphics is optimized for simple arithmetic (increment) on unsigned integers.

Check-list for OpenCL™ Optimizations

- Mapping Memory Objects
- Using Buffers and Images Appropriately
- Using Floating Point for Calculations
- Using Compiler Options for Optimizations
- Using Built-In Functions
- Loading and Storing Data in Greatest Chunks
- Applying Shared Local Memory
- Using Specialization in Branching
- Considering native_ half_ Versions of Math Built-Ins
- Using the Restrict Qualifier for Kernel Arguments
- Avoiding Handling Edge Conditions in Kernels

Mapping Memory Objects

Host code shares physical memory with both OpenCL™ devices: the CPU and the Intel® Graphics. So consider using combination of clEnqueueMapBuffer and clEnqueueUnmapBuffer instead of calls to clEnqueueReadBuffer or clEnqueueWriteBuffer. The recommendation applies to the CPU OpenCL device, Intel Graphics OpenCL device, and also to the shared (CPU and Intel Graphics devices) context.

Notice that there are two ways to ensure zero-copy path on memory objects mapping. The preferred way is to request the OpenCL runtime to allocate memory with CL_MEM_ALLOC_HOST_PTR, so it is originally mirrored on the host in the efficient way.

Another way is to allocate properly aligned and sized memory yourself and share the pointer with the OpenCL framework by using clCreateBuffer with the CL_MEM_USE_HOST_PTR flag. This is a viable option, if your application uses a specific memory management algorithm, or if you want to wrap existing native application memory allocations. The CL_MEM_USE_HOST_PTR flag enables your application to share its memory allocation directly with the OpenCL runtime implementation, and avoid memory copies of the buffer.

For efficiency reasons such a host-side pointer must be allocated for the conditions:

- The amount of memory you allocate and the size of the corresponding OpenCL™ buffer must be multiple of the cache line sizes (64 bytes).
- Always use 4k alignment (page alignment) when you allocate the host memory for sharing with OpenCL devices.

Consider the following pseudo-code example:

```
int cachelineSize = clGetDeviceInfo(device, …CL_DEVICE_GLOBAL_MEM_CACHELINE);//bytes
int arraySizeAligned =  cachelineSize*(1+(arraySize-1)/cachelineSize);//aligned
void* inputArray = _aligned_malloc(arraySizeAligned, 4096);
cl_mem inputBuf = clCreateBuffer(…CL_MEM_USE_HOST_PTR, arraySizeAligned, inputArray);
```

Similarly, page-align host pointers for the API calls that accept the pointers:

```
void* dstArray = _aligned_malloc(arraySize, 4096);
// example of reading a buffer back from Intel® Graphics device (single-device or shared
context), notice that clEnqueueMapBuffer is a better solution
clEnqueueReadBuffer(queue, buffer, FALSE, 0, arraySize, dstArray,0, NULL, NULL);
```

You can map image objects as well. For a context containing only the Intel Graphics device, the mapping of images is less efficient, since the images are tiled and cannot be mapped directly.

You can find additional details in the **See Also** section below.

See Also

Sharing Resources Efficiently

Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics

Using Buffers and Images Appropriately

On both CPU and Intel® Graphics devices, buffers usually perform better than images: more data transfers per read or write operation for buffers with much lower latency.

If your algorithm does not require linear data interpolation or specific border modes, consider using buffers instead of images. Still, if your legacy code uses images, or if you want to use the linear interpolation ability of the sampler, consider using the cl_khr_image2d_from_buffer extension which offers creating zero copy image aliases for the buffers.

To improve performance on the Intel Graphics, do the following:

- Consider using images if you need linear interpolation between pixel values.
- Consider using images for irregular access patterns. For example, use buffers when processing in memory (in row-major) order. Yet prefer image2D and texture sampling your access pattern is other than simple linear. For example, a kernel that reads diagonally or generally irregular positions.
- Use local memory for explicit caching of data values rather than relying on sampler's caches, as the caches do not support image *write* operations.
- Notice however, that for (even mildly large) look-up tables the regular global memory is preferable over local memory.
- Use constant samplers to be able to specify and optimize the sampling behavior in compile time.
- Consider using the CL_ADDRESS_CLAMP_NONE as it is the fastest addressing mode, and use CL_ADDRESS_CLAMP_TO_EDGE rather than CL_ADDRESS_CLAMP.

In general, image2D sampling on the Intel Graphics offers:

- Free type conversions. For example, uchar4 to uint4 (unavailable on CPU).
- Automatic handling image boundaries (slower on the CPU).
- Fast bilinear sampling (works slow on CPU; may vary between devices).

Notice that images are software-emulated on a CPU. So, make sure to choose the fastest interpolation mode that meets your needs. Specifically:

- Nearest-neighbor filtering works well for most (interpolating) kernels.

- Linear filtering might decrease CPU device performance.

See Also

Memory Access Overview

Applying Shared Local Memory (SLM)

Using Image2D From Buffer Extension

Using Floating Point for Calculations

Intel® Graphics device is much faster for floating-point add, sub, mul and so on in compare to the int type.

For example, consider the following code that performs calculations in type int4:

```
__kernel void amp (__constant uchar4* src, __global uchar4* dst)
        …
        uint4 tempSrc = convert_uint4(src[offset]);//Load one RGBA8 pixel
        //some processing
        uint4 value = (tempSrc.z + tempSrc.y + tempSrc.x);
        uint4 tempDst = value + (tempSrc - value) * nSaturation;
        //store
        dst[offset] = convert_uchar4(tempDst);
}
```

Below is its float4 equivalent:

```
__kernel void amp (__constant uchar4* src, __global uchar4* dst)
        …
        uint4 tempSrc = convert_uint4(src[offset]);//Load one RGBA8 pixel
        //some processing
        float4 value = (tempSrc.z + tempSrc.y + tempSrc.x);
        float4 tempDst = mad(tempSrc - value,  fSaturation, value);
        //store
        dst[offset] = convert_uchar4(tempDst);
}
```

Intel® Advanced Vector Extensions (Intel® AVX) support (if available) accelerates floating-point calculations on the modern CPUs, so floating-point data type is preferable for the CPU OpenCL device as well.

NOTE The compiler can perform automatic fusion of multiplies and additions. Use compiler flag -cl-mad-enable to enable this optimization when compiling for both Intel® Graphics and CPU devices. However, explicit use of the "mad" built-in ensures that it is mapped directly to the efficient instruction.

Using Compiler Options for Optimizations

The -cl-fast-relaxed-math compiler option is the most general and powerful among other performance related options. Notice that the option affects the compilation of the entire OpenCL program, so it does not permit fine control of the resulting numeric accuracy. You may want to consider experimenting with native_* equivalents separately for each specific built-in instead, keeping track of the resulting accuracy. Please find more details on this approach in the "Considering native_ and half_ versions of Math Built-Ins" section.

Refer to the *User Manual - OpenCL™ Code Builder* for the list of compiler options for the specific optimizations.

NOTE Intel® CPU and Intel® Graphics devices support different sets of options.

See Also

User Manual - OpenCL™ Code Builder

Considering native_ half_ Versions of Math Built-Ins

Using Built-In Functions

OpenCL™ software technology offers a library of built-in functions, including vector variants. Using the built-in functions is typically more efficient than implementing them manually in OpenCL code. For example, consider the following code example:

```
int tid = get_global_id(0);
c[tid] = 1/sqrt(a[tid] + b[tid]);
```

The following code uses the built-in rsqrt function to implement the same example more efficiently:

```
int tid = get_global_id(0);
c[tid] = rsqrt(a[tid] + b[tid]);
```

See other examples of simple expressions and built-ins based equivalents below:

```
dx * fCos + dy * fSin == dot( (float2)(dx, dy),(float2)(fCos, fSin))
x * a - b  == mad(x, a, -b)
sqrt(dot(x, y)) == distance(x,y)
```

The only exception is using mul24 as it involves redundant overflow-handling logic:

```
int iSize = x*y;//prefer general multiplication, not mul24(x,y);
```

Also use specialized built-in versions where possible. For example, when the x value for xy is ≥0, use powr instead of pow.

See Also

The OpenCL 2.0 C Specification at https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf

Loading and Storing Data in Greatest Chunks

"Saturating" the available memory bandwidth is very important. Bytes data types actually load integer data types (DWORDS), but also trigger instructions to pack and unpack data. Using (u)int4 or float4 for buffers saves a lot of compute, even if you unpack data manually afterward. In other words, you should avoid using uchar4 or char4. See the example below:

```
__kernel void amp (__constant uchar4* src, __global uchar4* dst)
        …
        uint4 tempSrc = convert_uint4(src[offset]);//Load one RGBA8 pixel
        …
        //some processing
        …
        dst[offset] = convert_uchar4(tempDst);
}
```

Consider data accesses by using int4 data type:

```
__kernel void amp (__constant uint4* src, __global uint4* dst)
        …
        uint4 tempSrc = src[offset]; // Load 4 RGBA8 pixels
        …
        //some processing in uint4
        uint r0 = (tempSrc.x & 0xff);//Red component of 1st pixel
        uint r1 = (tempSrc.y & 0xff);//Red component of 2nd pixel
        …
        tempSrc.x >>= 8;
        tempSrc.y >>= 8;
        …
        tempSrc.x >>= 8;
        tempSrc.y >>= 8;
        …
        uint a0 = (tempSrc.x & 0xff);// Alpha component of 1st pixel
        uint a1 = (tempSrc.y & 0xff);// Alpha component of 2nd pixel
        //any calculations on the individual components
        …
        uint4 final = 0; // repack them:
        final.x = (r0) | ((g0) << 8) | ((b0) << 16) | ((a0) << 16);//first pixel
        final.y = (r1) | ((g1) << 8) | ((b1) << 16) | ((a1) << 16);//second pixel
        …
        dst[offset] = final;
}
```

NOTE The global size is 1/4th of the original size in the second example above.

If your kernel operates on floating-point data, consider using float4 data type, which gets four times as much data in one load. It also helps to ensure that the kernel has enough work to do, amortizing the work-item scheduling overheads.

For the CPU device this optimization is equivalent to explicit (manual) vectorization, see the "Using Vector Data Types" section for more information.

Accessing data in greater chunks can improve the Intel® Graphics device data throughput, but it might slightly reduce the CPU device performance as also explained in the "Using Vector Data Types" section.

See Also

Using Vector Data Types

Applying Shared Local Memory

Intel® Graphics device supports the Shared Local Memory (SLM), attributed with __local in OpenCL™. This type of memory is well-suited for scatter operations that otherwise are directed to global memory. Copy small table buffers or any buffer data, which is frequently reused, to SLM. Refer to the "Local Memory Consideration" section for more information.

An obvious approach to populate SLM is using the for loop. However, this approach is inefficient because this code is executed for every single work-item:

```
__kernel void foo_SLM_BAD(global int * table,
                        local int * slmTable /*256 entries*/)
{
        //initialize shared local memory (performed for each work-item!)
        for( uint index = 0;  index < 256;  index ++ )
                slmTable[index] = table[index];
        barrier(CLK_LOCAL_MEM_FENCE);
```

The code copies the table over and over again, for every single work-item.

An alternative approach is to keep the for loop, but make it start at an index set by getting the local id of the current work-item. Also get the size of the work-group, and use it to increment through the table:

```
__kernel void foo_SLM_GOOD(global int * table,
                        local int * slmTable /*256 entries*/)
{
        //initialize  shared local memory
        int   lidx = get_local_id(0);
        int   size_x = get_local_size(0);
        for( uint   index = lidx; index < 256; index += size_x )
                slmTable[index] = table[index];
        barrier(CLK_LOCAL_MEM_FENCE);
```

You can further avoid the overhead of copying to SLM. Specifically for the cases, when number of SLM entries equals the number of work-items, every work-item can copy just one table entry. Consider populating SLM this way:

```
__kernel void foo_SLM_BEST(global int * table,
                        local int * slmTable)
{
        //initialize  shared local memory
        int   lidx = get_local_id(0);
        int   lidy = get_local_id(1);
        int   index = lidx + lidy * get_local_size(0);
        slmTable[index] = table[index]; barrier(CLK_LOCAL_MEM_FENCE);
```

If the table is smaller than the work-group size, you might use the "min" instruction. If the table is bigger, you might have several code lines that populate SLM at fixed offsets (which actually is unrolling of the original for loop). If the table size is not known in advance, you can use a realfor loop.

Applying SLM can improve the Intel Graphics data throughput considerably, but it might slightly reduce the performance of the CPU OpenCL device, so you can use a separate version of the kernel.

See Also

__local Memory

Using Specialization in Branching

You can improve the performance of both CPU and Intel® Graphics devices by converting the uniform conditions that are equal across all work-items into compile time branches, a techniques known as specialization.

The approach, which is sometimes referred as Uber-Shader in the pixel shader context, is to have a single kernel that implements all needed behaviors, and to let the host logic disable the paths that are not currently required. However, setting constants to branch on calculations wastes the device facilities, as the data is still being calculated before it is thrown away. Consider a preprocess approach instead, using #ifndef blocks.

Original kernel that uses constants to branch:

```
__kernel void foo(__constant int* src,
                 __global int*
dst,                                                       unsigned char
bFullFrame, unsigned char bAlpha)
{
        …
        if(bFullFrame)//uniform condition (equal for all work-items
        {
        …
                if(bAlpha) //uniform condition
                {
                …
                }
                else
                {
                …
                }
        else
        {
        …
        }
}
```

The same kernel with compile time branches:

```
__kernel void foo(__constant int* src,
                 __global int* dst)
{
        …
        #ifdef bFullFrame
        {
        …
                #ifdef bAlpha
                {
                …
                }
                #else
                {
                …
                }
                #endif
        #else
        {
        …
        }
        #endif
}
```

Also consider similar optimization for other constants.

Finally, avoid or minimize use of branching in short computations with using min, max, clamp or select built-ins instead of “if and else”.

Also, optimizing specifically for the OpenCL™ Intel Graphics device, ensure all conditionals are evaluated outside of code branches (for the CPU device it does not make any difference).

For example, the following code demonstrates conditional evaluation in the conditional blocks:

```
if(x && y || (z && functionCall(x, y, z))
 {
    // do something
 }
 else
 {
    // do something else
 }
```

The following code demonstrates the conditional evaluation moved outside of the conditional blocks:

```
//improves compilation time for Intel® Graphics device
bool comparison = x && y || (z && functionCall(x, y, z));
 if(comparison)
 {
    // do something
 }
 else
 {
    // do something else
 }
```

See Also

Using the Preprocessor for Constants

Considering native_ and half_ Versions of Math Built-Ins

OpenCL™ API offers two basic ways to trade precision for speed:

- native_* and half_* math built-ins, which have lower precision, but are faster than their un-prefixed variants
- Compiler optimization options that enable optimizations for floating-point arithmetic for the whole OpenCL program (for example, the -cl-fast-relaxed-math flag).

For the list of other compiler options and their description please refer to the *Intel® Code Builder for OpenCL™ API - User Manual*. In general, while the -cl-fast-relaxed-math flag is a quick way to get potentially large performance gains for kernels with many math operations, it does not permit fine control of numeric accuracy. Consider experimenting with native_* equivalents separately for each specific case, keeping track of the resulting accuracy.

The native_ versions of math built-ins are generally supported in hardware and run substantially faster, while offering lower accuracy. Use native trigonometry and transcendental functions, such as sin, cos, exp or log, when performance is more important than precision.

The list of functions that have optimized versions support is provided in "Working with cl-fast-relaxed-math Flag" section of the *OpenCL Code Builder - User's Guide*.

See Also

OpenCL™ Build and Linking Options chapter of the Intel® Code Builder for OpenCL™ API - User Manual

Using the Restrict Qualifier for Kernel Arguments

Consider using the restrict (defined by the C99) type qualifier for kernel arguments (pointers) in the kernel signature. The qualifier declares that pointers do not alias each other, which helps the compiler limit the effects of pointer aliasing, while aiding the caching optimizations.

```
__kernel void foo( __constant float* restrict a,
                          __constant float* restrict b,
                          __global float* restrict result)
```

NOTE You can use the restrict qualifier only with kernel arguments. In the specific example above, it enables the compiler to assume that pointers a, b, and result do point to the different locations. So you must ensure that the pointers do not point to overlapping memory regions.

Avoiding Handling Edge Conditions in Kernels

Consider this smoothing 2x2 filter:

```
__kernel void smooth(const __global float* input,
                     __global float* output)
{
  const int myX = get_global_id(0);
  const int myY = get_global_id(1);
  const int image_width = get_global_size(0);
  uint neighbors = 1;
  float sum = 0.0f;
  if ((myX + 1) < (image_width-1))
  {
    sum += input[myY * image_width + (myX + 1)];
    ++neighbors;
  }
  if (myX > 0)
  {
    sum += input[myY * image_width + (myX - 1)];
    ++neighbors;
  }
  if ((myY + 1) < (image_height-1))
  {
    sum += input[(myY + 1) * image_width + myX];
    ++neighbors;
  }
  if (myY > 0)
  {
    sum += input[(myY – 1) * image_width + myX];
    ++neighbors;
  }
  sum += input[myY * image_width + myX];
  output[myY * image_width + myX] = sum / (float)neighbors;
}
```

Assume that you have a full HD image with size of 1920x1080 pixels. The four edge if conditions are executed for every pixel, that is, roughly two million times.

However, they are only relevant for the 6000 pixels on the image edges, which make 0.2% of all the pixels. For the remaining 99.8% work-items, the edge condition check is a waste of time. Also compare how shorter and easier to perceive the following code, which does not perform any edge check:

```
__kernel void smooth(const __global float* input,
                     __global float* output)
{
  const int myX = get_global_id(0);
  const int myY = get_global_id(1);
  const int image_width = get_global_size(0);
  float sum = 0.0f;
  sum += input[myY * image_width + (myX + 1)];
  sum += input[myY * image_width + (myX - 1)];
  sum += input[(myY + 1) * image_width + myX];
  sum += input[(myY – 1) * image_width + myX];
  sum += input[myY * image_width + myX];
  output[myY * image_width + myX] = sum / 5.0f;
}
```

This code requires padding (enlarging) the input buffer appropriately, if using the original global size. This way querying the neighbors for the border pixels does not result in buffer overrun.

If padding through larger input is not possible, make sure you use the min and max built-in functions, so that checking a work-item does not access outside the actual image and adds only four lines:

```
__kernel void smooth(const __global float* input,
                     __global float* output)
{
  const int image_width = get_global_size(0);
  const int image_height = get_global_size(0);
  int myX = get_global_id(0);
  //since for myX== image_width-1 the (myX+1) is incorrect
 myX =  min(myX, image_width -2);
  //since for myX==0 the (myX-1) is incorrect
 myX =  max(myX, 1);
  int myY = get_global_id(1);
  //since for myY== image_height-1 the (myY+1) is incorrect
 myY =  min(myY, image_height -2);
  //since for myY==0 the (myY-1) is incorrect
 myY =  max(myY , 1);
  float sum = 0.0f;
  sum += input[myY * image_width + (myX + 1)];
  sum += input[myY * image_width + (myX - 1)];
  sum += input[(myY + 1) * image_width + myX];
  sum += input[(myY - 1) * image_width + myX];
  sum += input[myY * image_width + myX];
  output[myY * image_width + myX] = sum / 5.0f;
}
```

At a cost of duplicating calculations for border work-items this code avoids testing for the edge conditions, which is otherwise necessary to perform for the all work-items.

One more approach is to ignore the pixels on the edge, for example, by executing the kernel on a 1918x1078 sub-region within the buffer. OpenCL™ 1.2 and higher enables you to use global_work_offset parameter with clEnqueueNDRangeKernel to implement this behavior. However, use 1912 for first dimension of the global size, as 1918 is not a multiple of 8, which means potential underutilization of the SIMD units. Notice that OpenCL 2.0 offers "non-uniform work-groups" feature which handles global sizes that are not multiple of underlying SIMD in the very efficient way. Refer to the **See Also** section below for details.

NOTE Using image types along with the appropriate sampler (CL_ADDRESS_REPEAT or CLAMP) also automates edge condition checks for data reads. Refer to the "Using Buffers and Images Appropriately" section for pros and contras of this approach.

See Also

Using Buffers and Images Appropriately

OpenCL™ 2.0 Non-Uniform Work-Groups

Performance Debugging

- Host-Side Timing
- Profiling Operations Using OpenCL Profiling Events
- Comparing OpenCL Kernel Performance with Performance of Native Code
- Getting Credible Performance Numbers
- Using Tools

Host-Side Timing

The following code snippet is a host-side timing routine around a kernel call (error handling is omitted):

```
float start = …;//getting the first time-stamp
        clEnqueueNDRangeKernel(g_cmd_queue, …);
        clFinish(g_cmd_queue);// to make sure the kernel completed
float end = …;//getting the last time-stamp
float time = (end-start);
```

In this example, host-side timing is implemented using the following functions:

- clEnqueueNDRangeKernel adds a kernel to a queue and immediately returns
- clFinish explicitly indicates the completion of kernel execution. You can also use clWaitForEvents.

Wrapping the Right Set of Operations

When using any host-side routine for evaluating performance of your kernel, ensure you wrapped the proper set of operations.

For example, avoid potentially costly and/or serializing routine, like:

- Including various printf calls
- File input or output operations
- and so on

Also profile kernel execution and data transferring separately by using OpenCL™ profiling events. Similarly, keep track of compilation and general initialization costs, like buffer creation separately from the actual execution flow.

See Also

Profiling Operations Using OpenCL™ Profiling Events

Profiling Operations Using OpenCL™ Profiling Events

The following code snippet measures kernel execution using OpenCL™ profiling events (error handling is omitted):

```
g_cmd_queue = clCreateCommandQueue(…CL_QUEUE_PROFILING_ENABLE, NULL);
clEnqueueNDRangeKernel(g_cmd_queue,…, &perf_event);
clWaitForEvents(1, &perf_event);
cl_ulong start = 0, end = 0;
clGetEventProfilingInfo(perf_event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
clGetEventProfilingInfo(perf_event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
//END-START gives you hints on kind of "pure HW execution time"
//the resolution of the events is 1e-09 sec
g_NDRangePureExecTimeMs = (cl_double)(end - start)*(cl_double)(1e-06);
```

Important caveats:

- The queue should be enabled for profiling (CL_QUEUE_PROFILING_ENABLE property) at the time of creation.
- You need to explicitly synchronize the operation using clFinish() or clWaitForEvents. The reason is that device time counters for the profiled command, are associated with the specified event.

This way you can profile operations on both Memory Objects and Kernels. Refer to the OpenCL™ 1.2 Specification for the detailed description of profiling events.

NOTE The host-side wall-clock time might return different results. For the CPU the difference is typically negligible.

See Also

The OpenCL™ 1.2 Specification at http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Comparing OpenCL™ Kernel Performance with Performance of Native Code

When comparing OpenCL™ kernel performance with native code, for example, C or Intel® Streaming SIMD Extensions (Intel® SSE) intrinsic, make sure that both versions are as similar as possible:

- Wrap exactly the same set of operations.
- Do not include program build time in the kernel execution time. You can amortize this step by program precompilation (refer to clCreateProgramFromBinary).

- Track data transfers costs separately. Also, use data mapping when possible, since this is closer to the way a data is passed in a native code (by pointers). Refer to the “Mapping Memory Objects” section for more information.
- Ensure the working set is identical for native and OpenCL code. Similarly, for correct performance comparison, access patterns should be the same (for example, rows compared to columns).
- Demand the same accuracy. For example, rsqrt(x) is inherently of higher accuracy than the __mm_rsqrt_ps SSE intrinsic. To use the same accuracy in native code and OpenCL code, do one of the following:
  - Equip __mm_rsqrt_ps in your native code with a couple of additional Newton-Raphson iterations to match the precision of OpenCL rsqrt.
  - Use native_rsqrt in your OpenCL kernel, which maps to the rsqrtps instruction in the final assembly code.
  - Use the relaxed-math compilation flag to enable similar accuracy for the whole program. Similarly to rsqrt, there are relaxed versions for rcp, sqrt, etc. Refer to the *User Manual - OpenCL™ Code Builder* for the full list

See Also

Mapping Memory Objects

Considering native_ Versions of Math Built-Ins

User Manual - OpenCL™ Code Builder

Getting Credible Performance Numbers

Performance measurements are done on a large number of invocations of the same routine. Since the first iteration is almost always significantly slower than the subsequent ones, the minimum value for the execution time is usually used for final projections. Projections could also be made using other measures such as average or geometric mean of execution time.

An alternative to calling the kernel many times is to use a single “warm-up” run.

The warm-up run might be helpful for small or "lightweight" kernels, for example, the kernels with execution time less than 10 milliseconds. Specifically, it helps to amortize the following potential (one-time) costs:

- Bringing data to the cache
- “Lazy” object creation
- Delayed initializations
- Other costs incurred by the OpenCL™ runtime.

NOTE You need to build your performance conclusions on reproducible data. If the warm-up run does not help or execution time still varies, you can try running a large number of iterations and then average the results. For time values that range too much use geomean.

Consider the following:

- For bandwidth-limited kernels, which operate on the data that does not fit in the last-level cache, the ”warm-up” run does not have as much impact on the measurement.
- For a kernel with a small number of instructions executed over a small data set, make sure there is a sufficient number of iterations, so the kernel runs for at least 20 milliseconds.

Kernels that are very lightweight do not give reliable data, so making them artificially heavier could give you important insights into the hotspots. For example, you can add loop in the kernel, or replicate its heavy pieces.

Refer to the “OpenCL Optimizations Tutorial” SDK sample for code examples of performing the warm-up activities before starting performance measurement. You can download the sample from the Intel® SDK for OpenCL Applications website at intel.com/software/opencl/.

See Also

OpenCL Optimizations Tutorial SDK sample

User Manual - OpenCL™ Code Builder

Using Tools

Once you get reproducible performance numbers, you need to choose what to optimize first.

First, make sure your general application logic is sane. Refer to the Application-Level Optimizations chapter of this document.

OpenCL™ Code Builder offers a powerful set of Microsoft Visual Studio* and Eclipse* plug-ins for "Build/Debug/Profile" capabilities. Most important features it offers are:

- OpenCL debugging at the API level, so you can inspect a trace of your application for redundant copies, errors returned by OpenCL APIs, excessive sync, and so on.
- Also it offers rich features for kernel development in OpenCL language like offline OpenCL language compilation with cross hardware support, Low Level Virtual Machine (LLVM) and assembly language viewer.
- Finally, the tool features OpenCL kernels debugging and performance experimenting with running kernels on a specific device without writing a host code.

Intel®**Graphics Performance Analyzers** (Intel® GPA) is a set of tools, which enable you to analyze and optimize OpenCL execution (by inspecting hardware queues, DMA packets flow and basic hardware counters) and also rendering pipelines in your applications.

Second step is optimization of the most time-consuming OpenCL kernels. Your can perform simple static analysis yourself, for example: inspect kernel code with a focus on intensive use of heavy math built-ins, loops, and other potentially expensive things.

But when it comes to the tools-assisted analysis, **Intel® VTune™ Amplifier XE** is most powerful tool for OpenCL optimization, which enables you to fine-tune you code for optimal OpenCL CPU and Intel Graphics device performance, ensuring that hardware capabilities are fully utilized.

See Also

Application-Level Optimizations

Intel® Code Builder for OpenCL™ API - User Manual

Profiling OpenCL™ Applications with System Analyzer and Platform Analyzer

Using Multiple OpenCL™ Devices

Using Shared Context for Multiple OpenCL™ Devices

Intel OpenCL™ implementation features Common Runtime, which enables you to interface with the Intel® Graphics and the CPU devices using a single context.

You can create a "shared" context with both devices. Commands, resource sharing and synchronization instructions on the different devices should follow the OpenCL specification requirements.

The following is an example showing a specific way to create a shared context:

```
shared_context = clCreateContextFromType(prop, CL_DEVICE_TYPE_ALL, …);
```

In general avoid CL_DEVICE_TYPE_ALL. Proper way to create shared context is to provide the list of devices explicitly:

```
cl_device_id devices[2] = {cpuDeviceId , gpuDeviceId};
cl_context shared_context = clCreateContext(prop, 2, devices, …);
```

If you need a context with either CPU or GPU device, use CL_DEVICE_TYPE_CPU or CL_DEVICE_TYPE_GPU explicitly. In this case, the context you create is optimized for the target device.

NOTE Shared context does not imply any "shared queue". The OpenCL specification requires you to create a separate queue per device. See the dedicated "HDR Tone Mapping for Post Processing using OpenCL - Multi-Device Version" SDK sample for examples.

See Also

HDR Tone Mapping for Post Processing using OpenCL - Multi-Device Version

Sharing Resources Efficiently

Objects, allocated at the context level, are shared between devices in the context. For example, buffers and images are effectively shared by default. Other resources that are shared automatically across all devices, include program and kernel objects.

NOTE Shared memory objects cannot be written concurrently by different command queues. Use explicit synchronization of the write access with OpenCL™ synchronization objects, such as events. Consider using sub-buffers, which enables you to simultaneously write to the non-overlapping regions.

You can also avoid implicit copying when you share data with the host, as explained in the "Mapping Memory Objects" section.

NOTE To avoid potential inefficiencies, especially associated with improper alignment, use 4k alignment for the host pointers in scenarios when the Intel® Graphics device is involved. Also align the allocation sizes to the cache line boundaries (64 bytes). Refer to the "Mapping Memory Objects" section for more details.

See Also

Writing to a Shared Resource

Mapping Memory Objects

Synchronization Caveats

Similarly to the regular case of multiple queues within the same context, you can wait on event objects from CPU and GPU queue (error checking is omitted):

```
cl_event eventObjects[2];
//notice that kernel object itself can be the same (shared)
clEnqueueNDRangeKernel(gpu_queue, kernel, … &eventObjects[0]);
```

```
//other commands for the GPU queue
//…
//flushing queue to start execution on the Intel® Graphics in parallel to populating to the CPU
queue below
//notice it is NOT clFinish or clWaitForEvents to avoid serialization
clFlush(gpu_queue);//assuming NO RESOURCE or other DEPENDENCIES with CPU device
clEnqueueNDRangeKernel(cpu_queue, kernel, … &eventObjects[1]);
//other commands for the CPU queue
//…
//now let's flush second queue
clFlush(cpu_queue);
//now when both queues are flushed, let's wait for both kernels to complete
clWaitForEvents(2, eventObjects);
```

In this example the first queue is flushed without blocking and waiting for results. In case of blocking calls like clWaitForEvents and clFinish, the actions are serialized with respect to devices. The reason is that in this example the commands do not get into the (second) queue before clWaitForEvents andclFinish in the first queue return (assuming you are in the same thread).

For the example, when proper serialization is critical refer to the "Writing to a Shared Resource" section.

See Also

Writing to a Shared Resource

Writing to a Shared Resource

According to the OpenCL™ specification, you need to ensure that the commands that change the content of a shared memory object, complete in the previous command queue before the memory object is used by commands, executed in another command-queue. One way to achieve this is using events:

```
cl_event eventGuard;
cl_buffer bufferShared=clCreateBuffer(shared_context,CL_MEM_READ_WRITE…);
//Populating the buffer from the host, queue is regular in-order
clEnqueueWriteBuffer(cpu_queue, bufferShared,…);
//Setting the arguments and processing buffer with a kernel
SetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&bufferShared);
…
clEnqueueNDRangeKernel(cpu_queue, kernel, … &eventGuard);
//make sure the first device is done
clWaitForEvents(1, &eventGuard);
//alternatively you can use clFinish(cpu_queue) if in the same thread
…
//Now using buffer by second device
clEnqueueWriteBuffer(gpu_queue, bufferShared,…);
clEnqueueNDRangeKernel(gpu_queue, kernel, … &eventGuard);
…
```

If you want to write data (or output kernel results) to the same buffer *simultaneously* on two devices, use properly aligned, non-overlapping sub-buffers.

```
cl_buffer bufferShared = clCreateBuffer(shared_context, CL_MEM_ WRITE …);
//make sure alignment for the resp devices
cl_int gpu_align;
clGetDeviceInfo(gpuDeviceId, CL_DEVICE_MEM_BASE_ADDR_ALIGN,…&gpu_align);
gpu_align /= 8; //in bytes
//make sure that cpuPortion is properly aligned first!
cl_buffer_region cpuBufferRegion = { 0, cpuPortion};
cl_buffer_region gpuBufferRegion = { cpuPortion, theRest};
cl_buffer subbufferCPU = clCreateSubBuffer(bufferShared, 0,
        CL_BUFFER_CREATE_TYPE_REGION, &cpuBufferRegion, &err);
cl_buffer subbufferGPU = clCreateSubBuffer(bufferShared, 0,
        CL_BUFFER_CREATE_TYPE_REGION, &gpuBufferRegion, &err);
//now work with 2 sub-buffers on 2 devices simultaneously - (refer to the //prev. section)
..
//the sub-resources should be released properly
clReleaseMemObject(subbufferCPU);
clReleaseMemObject(subbufferGPU);
clReleaseMemObject(bufferShared);
```

See Also

The OpenCL™ 1.2 Specification at http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Partitioning the Work

Using multiple devices requires creating a separate queue for each device. This section describes potential strategies for work partition between the devices (command queues).

Assigning work statically (according to statically determined relative device speed) might result in lower overall performance. Consider allocating work according to the current load and speed of devices. The speed of a device can be affected by OS or driver scheduling decisions and by dynamic frequency scaling.

There are several approaches to the dynamic scheduling:

- Coarse-grain partitioning of the work between CPU and GPU devices:
  - Use the inter-frame load-balancing with the naturally independent data pieces like video frames or multiple image files to distribute them between different devices for processing. This approach minimizes scheduling overheads. However it requires a sufficiently large number of frames. It also might increase a burden to the shared resources, such as shared last-level cache and memory bandwidth.
  - Use the intra-frame load-balancing to split between the devices the data that is currently being processed. For example, if it is an input image, the CPU processes its first half, and the GPU processes the rest. The actual splitting ratio should be adjusted dynamically, based on how fast the devices complete the tasks. One specific approach is to keep some sort of performance history for the previous frames. Refer to the dedicated "HDR Tone Mapping for Post Processing using OpenCL - Multi-Device Version" SDK sample for an example.
- Fine-grain partitioning - partitioning into smaller parts that are requested by devices from the pool of remaining work. This partitioning method simulates a "shared queue". Faster devices request new input faster, resulting in automatic load balancing. The grain size must be large enough to amortize associated overheads from additional scheduling and kernel submission.

NOTE When deciding on how to split the data between devices, you should take into account the recommended local and global size granularity of each device. Use sub-resources when performing output to the shared resources by multiple devices.

You can also have a task-parallel scheduler. The approach requires understanding of both: task nature and device capabilities. For example, in the multi-kernel pipeline the first kernel runs on the CPU, which is good for the particular RNG algorithm; the second runs on the GPU, which is good for the specific type of heavy math, such as native_sin, native_cos. This way different pipeline stages are assigned to different devices. Such kind of partitioning may provide performance gain in some custom partitioning schemes, but in general the adaptability of this approach might be limited. For example, if you have just two kernels in the pipeline, you do not have many parameters to tweak, as the scheduler can run either kernel0 on CPU and kernel1 on GPU, or vice versa. It is important to minimize the time one device spends waiting for another to complete the task. One approach is to place a fixed size pool between producers and consumers. In a simple double-buffering scheme, the first buffer is processed, while the second is populated with new input data.

See Also

Writing to a Shared Resource

HDR Tone Mapping for Post Processing using OpenCL - Multi-Device Version

Keeping Kernel Sources the Same

It is often convenient to keep a kernel source same for different devices. On the other hand, it is often important to apply specific optimizations per device.

If you need separate versions of kernels, one way to keep the source code base same, is using the preprocessor to create CPU-specific or GPU-specific optimized versions of the kernels. You can run clBuildProgram twice on the same program object, once for CPU with some flag (compiler input) indicating the CPU version, the second time for GPU and corresponding compiler flags. Then, when you create two kernels with clCreateKernel, the runtime has two different versions for each kernel.

To maintain different versions of a kernel, consider using preprocessor directives over regular control flow, as explained in the “Using Specialization in Branching” section. Kernel prototype (the number of arguments and their types) should be the same for all kernels across all devices; otherwise you might get a CL_INVALID_KERNEL_DEFINITION error.

See Also

Mapping Memory Objects

Using Buffers and Images Appropriately

Using Floating Point for Calculations

Applying Shared Local Memory (SLM)

Notes on Branching/Loops

Considering native_ Versions of Math Built-Ins

Basic Frequency Considerations

Device performance can be affected by dynamic frequency scaling. For example, running long kernels on both devices simultaneously might eventually result in one or both devices stopping use of the Intel® Turbo Boost Technology. This might result in overall performance decrease even in compare to single-device scenario.

Similarly, in the single (Intel® Graphics) device scenario, a high interrupt rate and frequent synchronization with the host can raise the frequency of the CPU and drag the frequency of Intel® Graphics down. Using in-order queues can mitigate this.

See Also

Intel Turbo Boost Technology Support

Partitioning the Work

Avoiding Needless Synchronization

Eliminating Device Starvation

It is important to schedule command-queue for each device asynchronously. Host-queue multiple kernels first, then flush the queues so kernels begin executing on the devices, and finally wait for results. Refer to the Section "Synchronization Caveats" for more information.

Another approach is having a separate thread for GPU command-queue. Specifically, you can dedicate a physical CPU core for scheduling GPU tasks. To reserve a core, you can use the device fission extension, using which can prevent GPU starvation in some cases. Refer to the User Manual - OpenCL™ Code Builder for more information on the device fission extension.

Consider experimenting, as various trade-offs are possible.

See Also

Synchronization Caveats

User Manual - OpenCL™ Code Builder

Limitations of Shared Context with Respect to Extensions

Some potential implications of the shared context exist, for example, efficient zero-copy OpenGL* sharing is possible with the Intel® Graphics device, but makes you unable to create a shared context supporting this extension.

Refer to the *Intel® Code Builder for OpenCL™ API- User's Guide* for the specific extensions description and their behavior with respect to shared context.

See Also

Interoperability with Other APIs

OpenCL™ Code Builder – User's Guide

Coding for the Intel® CPU OpenCL™ Device

- Vectorization Basics for Intel® Architecture Processors
- Benefitting From Implicit Vectorization
- Vectorizer Knobs
- Using Vector Data Types
- Writing Kernels to Directly Target the Intel® Architecture Processors
- Work-Group Size Considerations
- Work-Group Level Parallelism

Vectorization Basics for Intel® Architecture Processors

Intel® Architecture Processors provide performance acceleration using Single Instruction Multiple Data (SIMD) instruction sets, which include:

- Intel Streaming SIMD Extensions (Intel SSE)
- Intel Advanced Vector Extensions (Intel AVX) instructions
- Intel Advanced Vector Extensions 2 (Intel AVX2) instructions

By processing multiple data elements in a single instruction, these ISA extensions enable data parallelism in scientific, engineering, or graphics applications.

When using SIMD instructions, vector registers hold group of data elements of the same data type, such as float or char. The number of data elements that fit in one register depends on the microarchitecture, and on the data type width, for example: starting with the 2nd Generation Intel Core™ Processors, the vector register width is 256 bits. Each vector (YMM) register can store eight float numbers, eight 32-bit integer numbers, and so on.

When using the SPMD technique, the OpenCL™ standard implementation can map the work-items to the hardware according to:

- Scalar code, when work-items execute one-by-one.

- SIMD elements, when several work-items fit in one register to run simultaneously.

The OpenCL Code Builder contains an implicit vectorization module, which implements the method with SIMD elements. Depending on the kernel code, this operation might have some limitations. If the vectorization module optimization is disabled, the SDK uses the method with scalar code.

See Also

Coding for the Intel® Architecture Processors

Benefitting From Implicit Vectorization

Benefitting From Implicit Vectorization

OpenCL™ Code Builder includes an implicit vectorization module as part of the program build process. When it is beneficial in terms of performance, this module packs several work-items together and executes them with SIMD instructions. This enables you to benefit from the vector units in the Intel® Architecture Processors without writing explicit vector code.

The vectorization module transforms scalar data type operations by adjacent work-items into an equivalent vector operations. When vector operations already exist in the kernel source code, the module scalarizes (breaks them down into component operations) and revectorizes them. This improves performance by transforming the memory access pattern of the kernel into a structure of arrays (SOA), which is often more cache-friendly than an array of structures (AOS).

You can find more details in the "Intel OpenCL™ Implicit Vectorization Module overview" article.

The implicit vectorization module works best for the kernels that operate on elements, which are four-byte wide, such as float or int data types. You can define the computational width of a kernel using the OpenCL vec_type_hint attribute.

Since the default computation width is four-byte, kernels are vectorized by default. If your kernel uses vectors explicitly, you can specify __attribute__((vec_type_hint(<typen>))) with typen of any vector type (for example, float3 or char4). This attribute indicates to the vectorization module that it should apply only transformations that are useful for this type.

The performance benefit from the vectorization module might be lower for the kernels that include a complex control flow.

To benefit from vectorization, your code does not need for loops within kernels. For best results, let the kernel deal with a single data element, and let the vectorization module take care of the rest. The more straightforward your OpenCL code is, the more optimization you get from vectorization.Writing the kernel in the plain scalar code is what works best for efficient vectorization. This method of coding avoids potential disadvantages associated with explicit (manual) vectorization described in the "Using Vector Data Types" section.

See Also

Vectorizer Knobs

Using Vector Data Types

Tips for Auto-Vectorization Module

Intel OpenCL™ Implicit Vectorization Module overview at http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf

Vectorizer Knobs

Several environment variables are related to vectorizer. The first one is CL_CONFIG_USE_VECTORIZER, which can be set to False and True respectively. Notice that just like any other environment variables this one affects the behavior of the vectorizer of the entire system (or shell instances) until variable gets unset explicitly (or shell(s) terminates). This specific variable affects code generation for Intel® Xeon Phi™ coprocessor device as well.

Another variable is CL_CONFIG_CPU_VECTORIZER_MODE (that affects code generation for CPU OpenCL device only). It effectively sets the vectorization “width” (when CL_CONFIG_USE_VECTORIZER = True):

- CL_CONFIG_CPU_VECTORIZER_MODE = 0 (default). The compiler makes heuristic decisions whether to vectorize each kernel, and if so which vector width to use.
- CL_CONFIG_CPU_VECTORIZER_MODE = 1. No vectorization by compiler. Explicit vector data types in kernels are left intact. This mode is the same as CL_CONFIG_USE_VECTORIZER = False.
- CL_CONFIG_CPU_VECTORIZER_MODE = 4. Disables heuristic and vectorizes to the width of 4.
- CL_CONFIG_CPU_VECTORIZER_MODE = 8. Disables heuristic and vectorizes to the width of 8.

NOTE Some kernels cannot be vectorized, so the vectorizer does not handle them, regardless the mode. Also be careful with manual overriding the compiler heuristic, build process fails if the target hardware doesn't support the specific vectorization width. Inspect the compiler output in the offline compiler tool (described in the product User’s Guide) on the messages related to vectorization.

See Also

User Manual - Intel® Code Builder for OpenCL™ API

Using Vector Data Types

To maximize CPU vector unit utilization, try to use vector data types in your kernel code. This technique enables you to map vector data types directly to the hardware vector registers. Thus, the data types used should match the width of the underlying SIMD instructions.

Consider the following recommendations:

- On the 2nd Generation Intel® Core™ Processors and higher with Intel® AVX support, use data types such as float8 or double4, so you bind code to the specific register width of the underlying hardware. This method provides maximum performance on a specific platform. However, performance on other platforms and supported Intel processors might be less than optimal.
- You may use wider data types, such as float16, to transparently cover many SIMD hardware register widths. However, using types wider than the underlying hardware is similar to loop unrolling. This method might improve performance in some cases, but also increases register pressure. Still consider using uchar16 data type to process four pixels simultaneously when operating on eight-bit-per-component pixels.
- When manually “vectorizing” an original kernel that uses scalar data types (like float) to use vector data types (like float8) instead, remember that each work-item processes N elements (for float/float8 example). Make sure you reduce the global size accordingly, so it is dividable by N.
- The int8 data type improves performance for the 4th Generation Intel® Core™ processors and higher.

Using this coding technique, you plan the vector-level parallelism yourself instead of relying on the implicit vectorization module (see the "Benefitting from Implicit Vectorization" section). This approach is useful in the following scenarios:

- You are porting code originally used Intel SSE/AVX/AVX2 instructions.
- You want to benefit from hand-tuned vectorization of your code.

The following example shows a multiplication kernel that targets the 256-bit vector units of the 2nd Generation Intel Core Processors:

```
__kernel __attribute__((vec_type_hint(float8)))
void edp_mul(__constant float8 *a,
           __constant float8 *b,
           __global float8 *result)
{
  int id = get_global_id(0);
  result[id] = a[id]* b[id];
}
```

In this example, the data passed to the kernel represents buffers of float8. The calculations are performed on eight elements together.

The attribute added before the kernel, signals the compiler, or the implementation that this kernel has an optimized vectorized form, so the implicit vectorization module does not operate on it. Use vec_type_hint to indicate to the compiler that your kernel already processes data using mostly vector types. For more details on this attribute, see the OpenCL™ 1.2 Specification.

See Also

Benefitting from Implicit Vectorization

The OpenCL™ 1.2 Specification at http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Writing Kernels to Directly Target the Intel® Architecture Processors

Using the OpenCL™ vector data types is a straightforward way to directly utilize the Intel® Architecture vector instruction set (see the "Using Vector Data Types" section). For instance, consider the following OpenCL standard snippet:

```
float4 a, b;
float4 c = a + b;
```

After compilation, it resembles the following C snippet in intrinsics:

```
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```

Or in assembly:

```
movaps xmm0, [a]
addps  xmm0, [b]
movaps [c], xmm0
```

However, in contrast to the code in intrinsics, an OpenCL kernel that uses the float4 data type, transparently benefits from Intel AVX if the compiler promotes float4 to float8. The vectorization module can pack work-items automatically, though it might be less efficient than manual packing.

If the native size for your kernel requires less than 128 bits and you want to benefit from explicit vectorization, consider packing work-items together manually.

For example, suppose your kernel uses the float2 vector type. It receives (x, y) float coordinates, and shifts them by (dx, dy):

```
__kernel void shift_by(__global float2* coords, __global float2* deltas)
{
  int tid = get_global_id(0);
  coords[tid] += deltas[tid];
}
```

To increase the kernel performance, you can manually pack pairs of work-items:

```
//Assuming the target is Intel® AVX enabled CPU
__kernel __attribute__((vec_type_hint(float8)))
void shift_by(__global float2* coords, __global float2* deltas)
{
  int tid = get_global_id(0);
  float8 my_coords = (float8)(coords[tid], coords[tid + 1],
                              coords[tid + 2], coords[tid + 3]);
  float8 my_deltas = (float8)(deltas[tid], deltas[tid + 1],
                              deltas[tid + 2] , deltas[tid + 3]);
  my_coords += my_deltas;
  vstore8(my_coords, tid, (__global float*)coords);
}
```

Every work-item in this kernel does four times as much work as a work-item in the previous kernel. Consequently, they require only one fourth the number of invocations, reducing the run-time overheads. However, when you use manual packing, you must also change the host code accordingly reducing the global size.

For vectors of 32-bit data types, such as int4, int8, float4 or float8, use explicit vectorization to improve the performance. Other data types (for example, char3) may cause an automatic upcast of the input data, which has a negative impact on performance.

For the best performance for a given data type, the vector width should match the underlying SIMD width. This value differs for different architectures. For example, consider querying the recommended vector width using clGetDeviceInfo with the CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT parameter. You get vector width of four for 2nd Generation Intel Core™ processors, but vector width of eight for higher versions of processors. So one viable option for vector width is using int8 so that the vector width fits both architectures. Similarly, for floating point data types, you can use float8 data to cover many potential architectures.

NOTE Using scalar data types such as int or float is often the most “scalable” way to help the compiler do right vectorization for the specific SIMD architecture.

See Also

Using Vector Data Types

Work-Group Size Considerations

It is recommended to let the OpenCL™ implementation automatically determine the optimal work-group size for a kernel: pass NULL for a pointer to the work-group size when calling clEnqueueNDRangeKernel.

If you want to experiment with work-group size, you need to consider the following:

- To get best performance from using the vectorization module (see the "Benefitting from Implicit Vectorization" section), the work-group size must be larger or a multiple of 8.
- To reduce the overhead of maintaining a workgroup, you should create work-groups that are as large as possible, which means 64 and more work-items. One upper bound is the size of the accessed data set as it is better not to exceed the size of the L1 cache in a single work group. Also there should be sufficient number of work-groups, see the "Work-Group Level Parallelism" section for more information.
- To accommodate multiple architectures, query the device for the CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE parameter by calling to clGetKernelWorkGroupInfo, and set the work-group size accordingly.
- If your kernel code contains the barrier instruction, the issue of work-group size becomes a tradeoff. The more local and private memory each work-item in the work-group requires, the smaller the optimal work-group size is. The reason is that a barrier also issues copy instructions for the total amount of private and local memory used by all work-items in the work-group in the work-group since the state of each work-item that arrived at the barrier is saved before proceeding with another work-item.

See Also

Work-Group Level Parallelism

Benefitting from Implicit Vectorization

Work-Group Level Parallelism

Since work-groups are independent, they can execute concurrently on different hardware threads. So the number of work-groups should be not less than the number of logical cores. A larger number of work-groups results in more flexibility in scheduling, at the cost of task-switching overhead.

Notice that multiple cores of a CPU as well as multiple CPUs (in a multi-socket machine) constitute a single OpenCL device. Separate cores are compute units. The Device Fission extension enables you to control compute unit utilization within a compute device. You can find more information on the Device Fission in the Intel® Code Builder for OpenCL™ API - User Manual.

For the best performance and parallelism between work-groups, ensure that execution of a work-group takes at least 100,000 clocks. A smaller value increases the proportion of switching overhead compared to actual work.

OpenCL™ Kernel Development for Intel® CPU OpenCL™ device

Why Optimizing Kernel Code Is Important?

An issued kernel is called many times by the OpenCL™ run-time. Therefore optimizing the kernel can bring a substantional benefit. If you move something out of the innermost loop in a typical native code, move it from the kernel as well. For example:

- Edge detection
- Constant branches
- Variable initialization
- Variable casts

Avoid Spurious Operations in Kernel Code

Since every line in kernel code is executed many times, make sure you have no spurious instructions in your kernel code.

Spurious instructions are not always obvious. Consider the following kernel:

```
__kernel void foo(const __global int* data, const uint dataSize)
{
  size_t tid = get_global_id(0);
  size_t gridSize = get_global_size(0);
  size_t workPerItem = dataSize / gridSize;
  size_t myStart = tid * workPerItem;
  for (size_t i = myStart; i < myStart + workPerItem; ++i)
  {
    //actual work
  }
}
```

In this kernel, the for loop is used to reduce the number of work-items and the overhead of keeping them. In this example every work-item recalculates the limit to the index i, but this number is identical for all work-items. Since the sizes of the dataset and the NDRange dimensions are known before the kernel launch, consider calculating the amount of work per item on the host once, and then pass the result as constant parameter.

In addition, using size_t for indices makes vectorization of indexing arithmetic less efficient. To improve performance, when your index fits the 32-bit integer range, use int data type, as shown in the following example:

```
__kernel void foo(const __global int* data, const uint workPerItem)
{
  int tid = get_global_id(0);
  int gridSize = get_global_size(0);
  //int workPerItem = dataSize / gridSize;
  int myStart = tid * workPerItem;
  for (int i = myStart; i < mystart + workPerItem; ++i)
```

Perform Initialization in a Separate Task

Consider the following code snippet:

```
__kernel void something(const __global int* data)
{
  int tid = get_global_id(0);
  if (0 == tid)
  {
    //Do some one-shot work
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
  //Regular kernel code
}
```

In this example, all work-items encounter the first branch, while the branch is relevant to only one of them. A better solution is to move the initialization phase outside the kernel code, either to a separate kernel, or to the host code.

If you need to run some kernel only once (for a single work-item), use clEnqueueTask, which is specially crafted for this purpose.

Use Preprocessor for Constants

Consider the following kernel:

```
__kernel void exponentor(__global int* data, const uint exponent)
{
  int tid = get_global_id(0);
  int base = data[tid];
  for (int i = 1; i < exponent; ++i)
  {
    data[tid] *= base;
  }
}
```

The number of iterations for the inner for loop is determined at runtime, after the kernel is issued for execution. However, you can use OpenCL™ dynamic compilation feature to ensure the exponent is known at kernel compile time, which is done during the host run time. In this case, the kernel appears as follows:

```
__kernel void exponentor(__global int* data)
{
  int tid = get_global_id(0);
  int base = data[tid];
  for (int i = 1; i < EXPONENT; ++i)
  {
    data[tid] *= base;
  }
}
```

The capitalization indicates that EXPONENT is a preprocessor macro.

The original version of the host code passes exponent_val through kernel arguments as follows:

```
clSetKernelArg(kernel, 1, exponent_val);
```

The updated version uses a compilation step:

```
sprintf(buildOptions, "-DEXPONENT=%u", exponent_val);
clBuildProgram(program, <...>, buildOptions, <...>);
```

Thus, the value of the EXPONENT is passed during preprocessing of the kernel code. Besides saving stack space used by the kernel, this also enables the compiler to perform optimizations, such as loop unrolling or elimination.

NOTE This approach requires recompiling the program every time the value of exponent_val changes. If you expect to change this value often, this approach is not advised. However, this technique is often useful for transferring parameters like image dimensions to video-processing kernels, where the value is only known at host run time, but does not change once it is defined.

Use Signed Integer Data Types

Many image-processing kernels operate on uchar input. To avoid overflows, you can convert 8-bit input values and process as 16- or 32-bit integer values. Use signed data types (shorts and ints) in both cases, if you need to convert to floating point and back.

Use Row-Wise Data Accesses

OpenCL™ enables you to submit kernels on one-, two- or three-dimensional index space. Consider using one-dimensional ranges for cache locality and to save index computations.

If a two- or three-dimensional range naturally fits your data dimensions, try to keep work-items scanning along rows, not columns. For example:

```
__kernel void smooth(const __global float* input,
                     uint image_width, uint image_height,
                     __global float* output)
{
  int myX = get_global_id(
0);
  int myY = get_global_id(
1);
  int myPixel = myY * image_width + myX;
  float data = input[myPixel];
  …
}
```

In the example above, the first dimension is the image width and the second is the image height. The following code is less effective:

```
__kernel void smooth(const __global float* input,
                     uint image_width, uint image_height,
                     __global float* output)
```

```
{
  int myY = get_global_id(
0);
  int myX = get_global_id(
1);
  int myPixel = myY * image_width + myX;
  float data = input[myPixel];
  …
}
```

In the second code example, the image height is the first dimension and the image width is the second dimension. The resulting column-wise data access is inefficient, since CPU OpenCL™ framework initially iterates over the first dimension.

The same rule applies if each work-item calculates several elements. To optimize performance, make sure work-items read from consecutive memory addresses.

Tips for Auto-Vectorization

Upon kernel compilation, the vectorization module often transforms the kernel memory access pattern from array of structures (AOS) to structure of arrays (SOA), which is SIMD friendly.

This transformation comes with a certain cost, specifically the transpose penalty. If you organize the input data in SOA instead of AOS, it reduces the transpose penalty.

For example, the following code suffers from transpose penalty:

```
__kernel void sum(__global float4* input, __global float* output)
{
int tid  = get_global_id(0);
output[tid] = input[tid].x + input[tid].y + input[tid].z + input[tid].w;
}
```

While the following piece of code does not suffer from the transpose penalty:

```
__kernel void sum(__global float* inx, __global float* iny, __global float* inz, __global float*
inw,  __global float* output)
{
int tid  = get_global_id(0);
output[tid] = inx[tid] + iny[tid] + inz[tid] + inw[tid];
}
```

Take care when dealing with branches. Particularly, avoid data loads and stores within the statements:

```
if (…) {//condition
        x = A[i1];// reading from A
        … // calculations
        B[i2] = y;// storing into B
} else {
         q = A[i1];// reading from A with same index as in first clause
         …  // different calculations
         B[i2] = w; // storing into B with same index as in first clause
}
```

The following code avoids loading from and storing to memory within branches:

```
temp1 = A[i1]; //reading from A in advance
if (…) {//condition
        x = temp1;
        … // some calculations
        temp2 = y; //storing into temporary variable
} else {
        q = temp1;
        … //some calculations
        temp2 = w; //storing into temporary variable
}
B[i2] =temp2; //storing to B once
```

See Also

Benefitting from Implicit Vectorization

Local Memory Usage

One typical GPU-targeted optimization uses local memory for caching of intermediate results. For CPU, all OpenCL™ memory objects are cached by hardware, so explicit caching by use of local memory just introduces unnecessary (moderate) overhead.

Avoid Extracting Vector Components

Consider the following kernel:

```
__constant float4 oneVec = (float4)(1.0f, 1.0f, 1.0f, 1.0f);
__kernel __attribute__((vec_type_hint(float4)))
void inverter2(__global float4* input, __global float4* output)
{
  int tid = get_global_id(0);
  output[tid] = oneVec - input[tid];
  output[tid].w = input[tid].w;
  output[tid] = sqrt(output[tid]);
}
```

For this example of the explicit vector code, extraction of the w component is very costly. The reason is that the next vector operation forces re-loading the same vector from memory. Consider loading a vector once and performing all changes, even to a single component, by use of vector operations.

In this specific case, two changes are required:

1. Modify the oneVec so that its w component is zero, causing only a sign change in the w component of the input vector.
2. Use float representation to manually change the sign bit of the w component back.

As a result, the kernel appears as follows:

```
__constant float4 oneVec = (float4)(1.0f, 1.0f, 1.0f, 0.0f);
__constant int4 signChanger = (int4)(0, 0, 0, 0x80000000);
__kernel __attribute__((vec_type_hint(float4)))
void inverter3(__global float4* input, __global float4* output)
{
  int tid  = get_global_id(0);
  output[tid] = oneVec - input[tid];
  output[tid] = as_float4(as_int4(output[tid]) ^ signChanger);
  output[tid] = sqrt(output[tid]);
}
```

At the cost of another constant vector, this implementation performs all the required operations addressing only full vectors. All the computations can be performed in float8.

Task-Parallel Programming Model Hints

Task-parallel programming model is general-purpose. It enables you to express parallelism by enqueuing multiple tasks. You can apply this model in the following scenarios:

- Performing different tasks concurrently by multiple threads. If you use this scenario, choose sufficient granularity of the tasks to enable good load balancing.
- Adding an extra queue (beside the conventional data-parallel pipeline) for tasks that occur less frequently and asynchronously, such as some scheduled events.

If your tasks are independent, consider using Out-of-Order queue.