Technology Guide

Advanced Encryption Standard Galois Counter Mode - Optimized GHASH Function

Authors

Erdinc Ozturk

Tomasz Kantecki

Kirk Yap

1 Introduction

Galois/Counter Mode (GCM) has two main components: encryption and authentication. Data is encrypted via Advanced Encryption Standard (AES) block cipher and an authentication tag is generated by applying a hash function (GHASH) to the ciphertext. There has been a tremendous amount of effort on improving the GCM performance over the years, in many fronts. Intel introduced and perfected the Intel® AES New Instructions (Intel® AES-NI), which includes both AES and PCLMULQDQ family of instructions. AES and GHASH components of GCM are highly parallelizable, and the current best implementations present similar performance for both AES and GHASH. Latency of the GCM operation is on par with the latency of the slower of these components. For example, assume GHASH latency is X cycles and AES latency is Y cycles, on an arbitrary length data. If X>Y, GCM latency is very close to X, (if a parallel implementation technique, such as function stitching, is used).

In this paper, we introduce two novel techniques to improve the GHASH performance. These techniques improve the latency of overall GHASH operation, and they can be utilized in any setting for GCM implementation. For proof of concept, we utilized these techniques on software implementations. Intel® Advanced Vector Extensions 512 (Intel AVX-512) allow parallel computations of AES and GHASH components, and high throughputs can be achieved in a single-buffer SIMD setting. We applied our techniques on this setting and achieved close to ~40% improvement for 64 byte message size and ~10% improvement for 16,384 byte message size.

This document is part of the Network & Edge Platform.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	August 2023	Initial release.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
GCM	Galois/Counter Mode
GHASH	Hash function over GF(2^{128}) used for constructing a Message Authentication Code (MAC) in the AES-GCM authenticated encryption cipher
SIMD	Single Instruction/Multiple Data
TLS	Transport Layer Security
VPN	Virtual Private Network
QUIC	Quick UDP Internet Connection

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
3rd Gen Intel® Xeon® Scalable Processor - Achieving 1 Tbps IPsec with Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Technology Guide	https://networkbuilders.intel.com/solutionslibrary/3rd-generation-intel-xeon-scalable-processor-achieving-1-tbps-ipsec-with-intel-advanced-vector-extensions-512-technology-guide
Intel® Advanced Vector Extensions 512	https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html
Intel® AVX-512 - High Performance IPsec with Intel® Xeon® Scalable Processor Technology Guide	https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-high-performance-ipsec-with-4th-gen-intel-xeon-scalable-processor-technology-guide
Intel® Multi-Buffer Crypto for IPsec Library	https://github.com/intel/intel-ipsec-mb
The Galois/Counter Mode of Operation (GCM) (D. A. McGrew and J. Viega)	https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
E. Ozturk and V. Gopal, "Enabling High-Performance Galois-Counter-Mode on Intel® Architecture Processors," 2012	https://www.intel.cn/content/dam/www/public/us/en/documents/software-support/enabling-high-performance-gcm.pdf
V. Gopal, W. Feghali, J. Guilford, E. Ozturk, G. Wolrich, M. Dixon, M. Locktyukhin and M. Perminov, "Fast Cryptographic Computation on Intel® Architecture Processors Via Function Stitching," 2010	https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-cryptographic-paper.pdf

2 Overview

2.1 Carry-less Multiplication

GHASH is defined over arithmetic in GF(2), and the operation known commonly as "carry-less multiplication" is multiplication of two polynomials with coefficients in GF(2) (Galois Field). In GF(2), every digit $a_i \in \{0,1\}$ and operations are realized modulo 2. Since there is no carry propagation, this operation is commonly known as "carry-less multiplication".

Arithmetic in GF(2): As can be seen from the truth table below, multiplication of two digits in GF(2) can be realized with a simple AND operation. Also, addition and subtraction of two digits in GF(2) can be realized with a simple XOR operation.

Table 3. Arithmetic in GF(2)

a	b	a*b	a+b	a-b
0	0	0*0 mod 2 = 0	0+0 mod 2 = 0	0-0 mod 2 = 0
0	1	0*1 mod 2 = 0	0+1 mod 2 = 1	0-1 mod 2 = 1
1	0	1*0 mod 2 = 0	1+0 mod 2 = 1	1-0 mod 2 = 1
1	1	1*1 mod 2 = 1	1+1 mod 2 = 0	1-1 mod 2 = 0

64-bit multiplication operation in GF(2) can be defined as:

```
GFMUL64(X,Y)=X*Y
```

Where, X and Y are 64-bit numbers representing degree-63 polynomials in GF(2) and the result is a 127-bit number representing a degree-126 polynomial in GF(2). The `GFMUL64(X,Y)` function is utilized in Intel® architectures, via the `PCLMULQDQ` family of instructions.

```
Inputs: X,Y
Output: Z=GFMUL64(X,Y)
Z=0
for i from 0 to 63:
       Z=Z^((X<<i)&(Y[i]))
```

2.2 Technology Description

2.2.1 Optimized Multiplication in GF(2^{128})

In this section, we introduce one of our novel techniques to improve overall GHASH performance. With this technique, the performance of multiplication in GF(2^{128}) is improved, which directly affects the overall GHASH performance.

The main building block of GHASH is multiplication in the field GF(2^{128}), which is defined by the polynomial: `P(x) =` $x^{128}+x^7+x^2+x+1$. The `RES = A*B mod P` operation can be defined as follows:

Table 4. GHASH Algorithm

Inputs:	A,B∈GF(2^{128}) P(x)=$x^{128}+x^7+x^2+x+1$.
Output:	RES=A*B mod P
Step 1:	Compute C=A*B
Step 2:	Compute RES=C mod P

This operation is depicted in Figure 1. All polynomials in GF(2^{128}) are represented as 128-bit integers. For a proof-of-concept implementation, we will present a software implementation. Therefore, every integer is represented with 64-bit digits. For example, `C[1:0]` represents two least significant digits and `C[3:2]` represents two most significant digits of the 256-bit integer C. It should be noted that in GF(2), the addition operation is realized with a simple XOR operation.

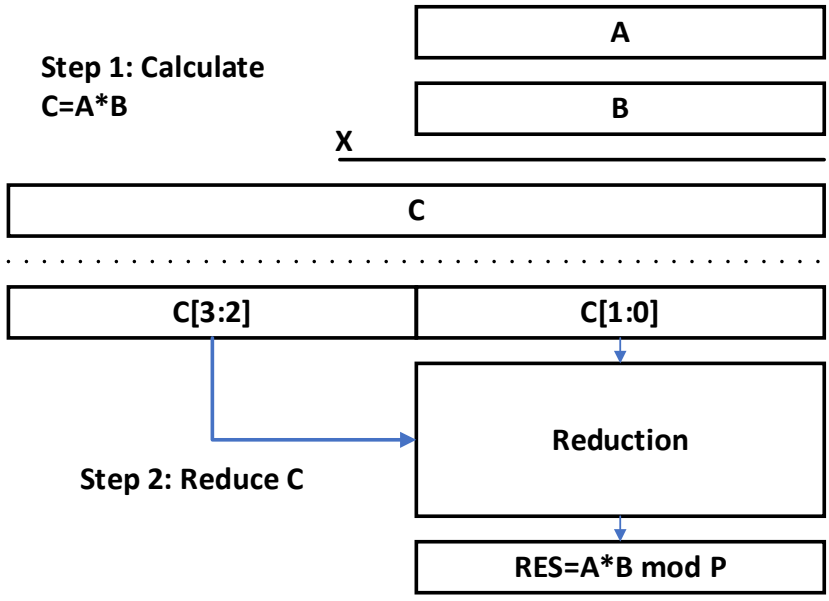

Figure 1. Simple Multiply-Reduce Algorithm

There are many well-known methods for Steps 1 and 2 of this multiplication operation. Karatsuba can be utilized for Step 1 and reduction techniques (Barrett, folding, shift-reduce, etc.) can be utilized for Step 2. A software-friendly version of the multiply-reduce routine can be defined as follows:

Table 5. Software Friendly GHASH Algorithm

Inputs:	A,B∈GF(2^{128}) Q(x)=x^7+x^2+x+1 (=x^{128} mod P)
Output:	RES=A*B mod P
Step 1:	**Compute C=A*B** C=GFMUL64(A[0],B[0])+ GFMUL64(A[0],B[1])*x^{64}+ GFMUL64(A[1],B[0])*x^{64}+ GFMUL64(A[1],B[1])*x^{128}
Step 2:	**Compute RES=C mod P** **Folding step 1:** C=C[3]*x^{192}+C[2:0] Y=C mod P =GFMUL64(C[3],Q)*x^{64}+C[2:0] **Folding step 2:** Y=Y[2]*x^{128}+Y[1:0] RES=Y mod P =GFMUL64(Y[2],Q)+Y[1:0]

This algorithm is depicted in Figure 2.

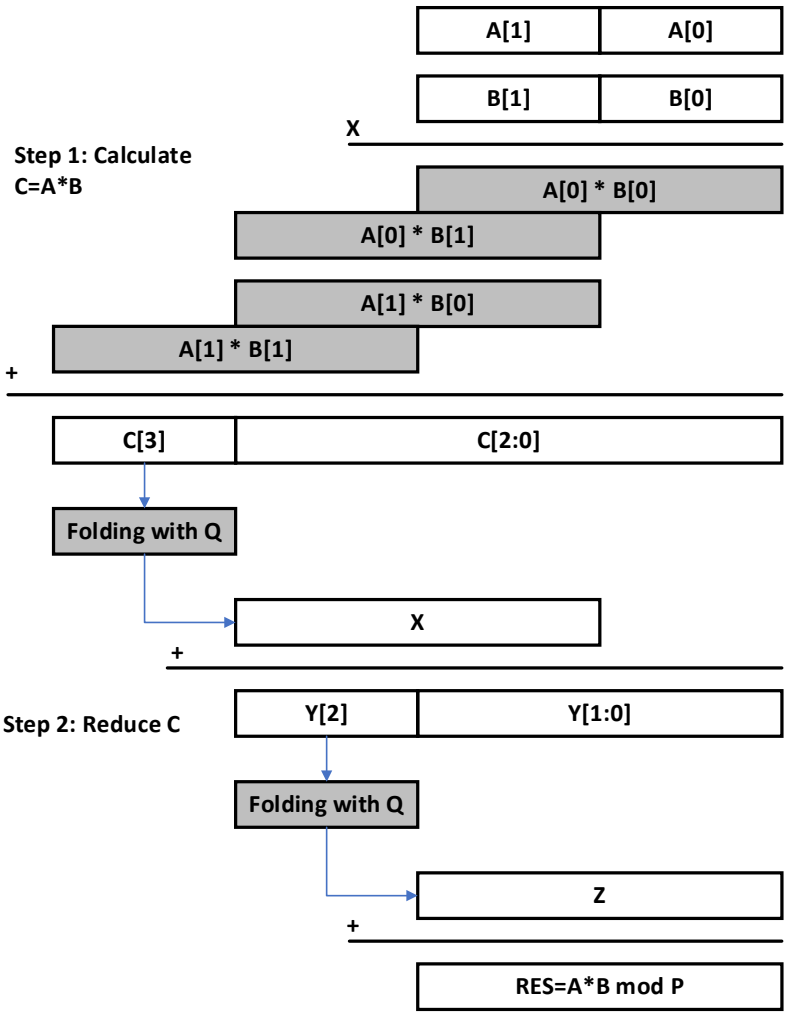

Figure 2. Software Friendly Multiply-Reduce Routine with Folding Approach

As can be seen in Figure 2, multiplication (Step 1) part of the operation can be realized with four independent 64-bit multiply operations, which can be realized via four PCLMULQDQ instructions. Reduction (Step 2) consists of two serial operations. To remove one of the serial steps, we modify the multiplication operation entirely with a precomputation step.

The following operation constitutes the main building block of GHASH where multiplication in GF(2^{128}) is denoted as *:

$X_i = (X_{i-1} \oplus C_i) * H$

In the context of GHASH, one of the inputs of this multiplication in GF(2^{128}) routine is always a variant of H, which is precomputed and stored before the GHASH operation is realized. We utilize this property in order to optimize the latency of GHASH operation.

First, we define a precomputed value K:

K=B[1]*Q

K is a value that is generated from the B input of the Multiplication in GF(2^{128}). K can be precomputed with the precomputed H table and the B input of the multiplication in GF(2^{128}) routine can always be selected as the input from the H table.

K (K[1:0]) is utilized in the multiplication operation instead of B[1]:

A[1]*B[1]*x^{128}≡A[1]*B[1]*Q mod P=A[1]*K

=GFMUL64(A[1], K[0]) + GFMUL64(A[1], K[1])*x^{64}

Step 1 of the algorithm is modified as follows:

Table 6. Improved GHASH Algorithm Step 1

Step 1:	**Compute C=A*B** Y=GFMUL64(A[0],B[0])+ GFMUL64(A[0],B[1])*x^{64}+ GFMUL64(A[1],B[0])*x^{64}+ GFMUL64(A[1],K[0])+ GFMUL64(A[1],K[1])*x^{64}

With this approach, multiply-reduce routine is optimized by removing a folding step from reduction. This is depicted in Figure 3. As can be seen in the figure, instead of the A[1]*B[1] 64-bit multiplication operation, we realize two 64-bit multiplication operations.

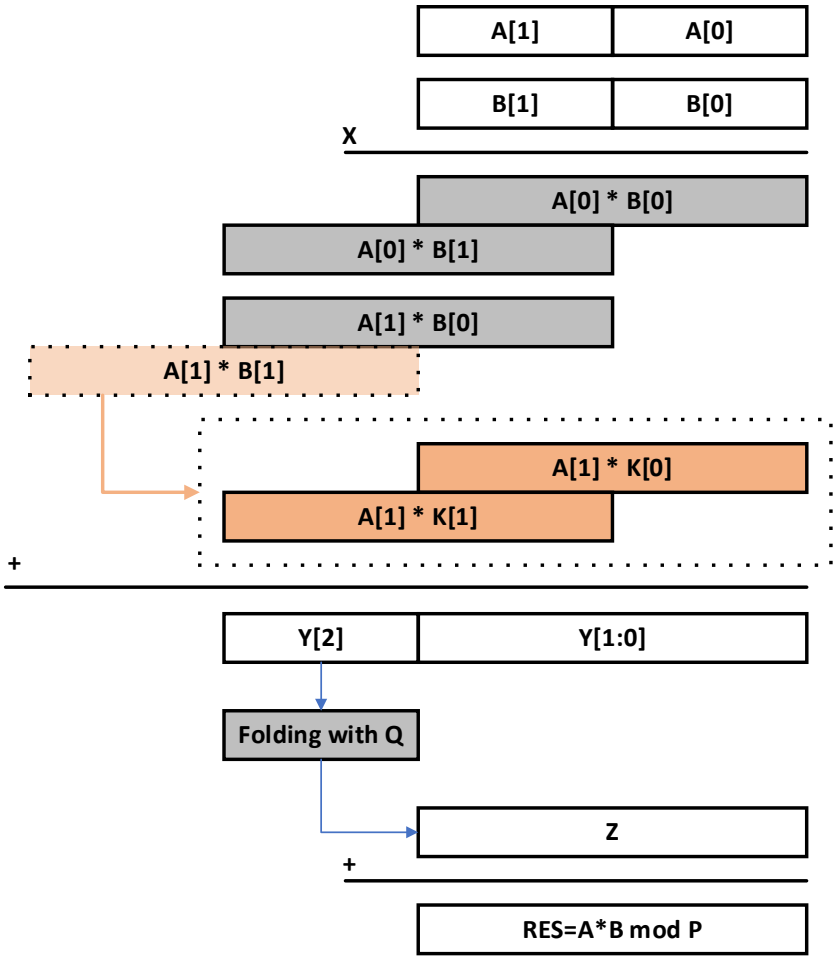

Figure 3. Optimized Multiply-Reduce Routine

Step 1 of the algorithm can be optimized with a slight modification. Since:

$GFMUL64(A[1],B[0])*x^{64}+GFMUL64(A[1],K[0])*x^{64}=GFMUL64(A[1],(K[0]+B[0]))*x^{64}$

We modify K as follows:

$K=B[1]*Q+B[0]*x^{64}$

Step 1 of the algorithm is further modified as follows:

Table 7. Optimized GHASH Algorithm Step 1

Step 1:	**Compute C=A*B** Y=GFMUL64(A[0],B[0])+ GFMUL64(A[0],B[1])*x^{64}+ GFMUL64(A[1],K[0])+ GFMUL64(A[1],K[1])*x^{64}

Our final optimized algorithm is detailed below and is detailed in Figure 4.

Table 8. Optimized GHASH Algorithm

Inputs:	A,B∈GF(2^{128}) Input: Precomputed K=B[1]*Q+B[0]*x^{64} Input: Q(x)=x^7+x^2+x+1(=x^{128} mod P)
Output:	RES=A*B mod P
Step 1:	**Compute Y** Y=GFMUL64(A[0],B[0])+ GFMUL64(A[0],B[1])*x^{64}+ GFMUL64(A[1],K[0])+ GFMUL64(A[1],K[1])*x^{64}
Step 2:	**Compute RES=Y mod P** **Folding step:** Y=Y[2]*x^{128}+Y[1:0] RES=Y mod P =GFMUL64(Y[2],Q)+Y[1:0]

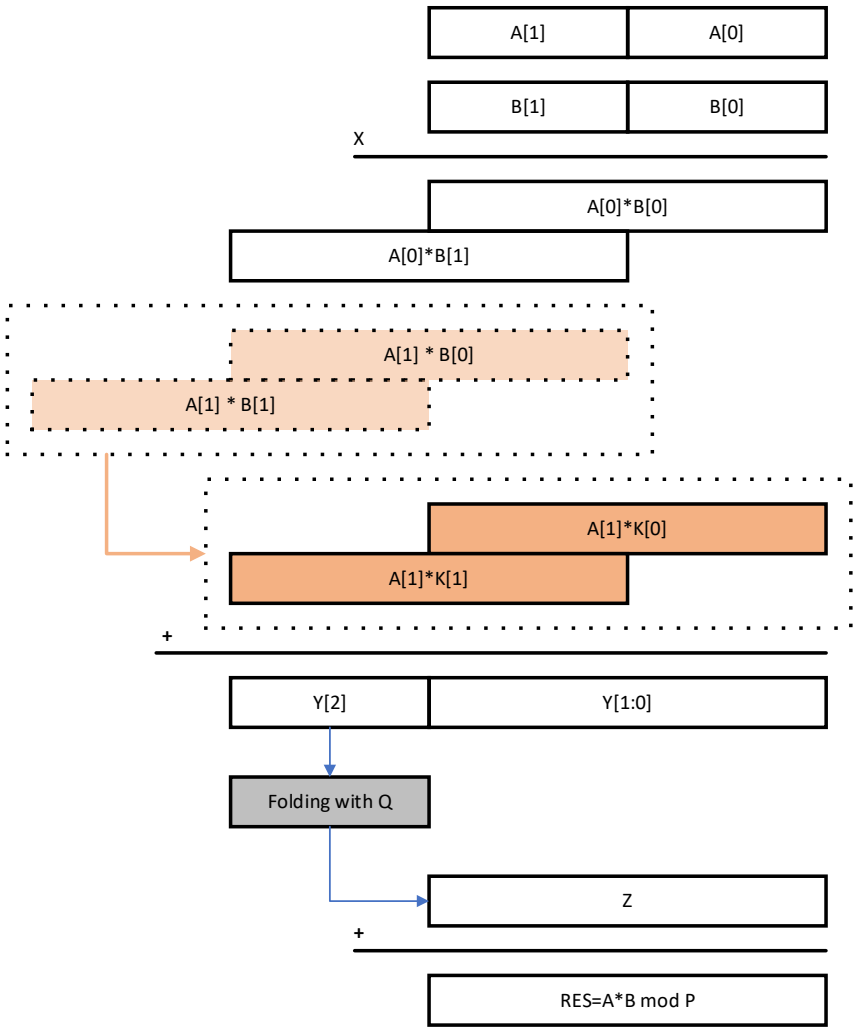

Figure 4. Further Optimized Multiply-Reduce Routine

2.2.2 Bit-Reflected Multiplication in GF(2^{128})

GHASH operation is defined over bit-reflected operands. GCM standard specifies that all operands are bit-reflected for GHASH computations. For software implementations, bit-reflection operation can be expensive. There are well-known methods that allow the use of PCLMULQDQ instructions without any bit-reflection operations. PCLMULQDQ instruction can be utilized with some extra precomputation steps to eliminate bit-reflection. Here, we first analyze the core multiplication operation.

We define a bit-reflection function:

$$bitreflect_i(X)=X'$$

where input X is treated as an i-bit value and bit-reflected. We have already defined the $GFMUL64(X,Y)=X*Y$ operation as the basic building block of PCLMULQDQ family of instructions. As stated, for GHASH, we utilize bit-reflected inputs.

Assume:

$$X' = bitreflect_{64}(X)$$

$$Y' = bitreflect_{64}(Y)$$

Then:

$$GFMUL64(X',Y') = bitreflect_{128}(Z<<1)$$

where:

$$Z = GFMUL64(X, Y)$$

We construct our 128-bit multiplier, named GFMUL128, using this building block, to realize $RES' = bitreflect_{128}(A*B \bmod P)$ operation. This operation is realized with the following algorithm, which is depicted in Figure 5.

Table 9. Bit Reflected GHASH Algorithm

Inputs:	$A'=bitreflect_{128}(A)$ $B'=bitreflect_{128}(B)$ $Q'=bitreflect_{64}(Q>>1)$ $(Q=x^{128} \bmod P)$
Output:	$RES'=A'*B' \bmod P'$
Step 1:	**Compute C'=A'*B'** where $C'=bitreflect_{128}(C<<1)$ $C'=GFMUL64(A'[0],B'[0])+$ $GFMUL64(A'[0],B'[1])*x^{64}+$ $GFMUL64(A'[1],B'[0])*x^{64}+$ $GFMUL64(A'[1],B'[1])*x^{128}$
Step 2:	**Compute RES=C mod P** **Folding step 1:** $Y=GFMUL64(C'[0],Q')+$ $C'[0]*x^{64}+C'[3:1]$ **Folding step 2:** $RES'=GFMUL64(Y[0],Q')+$ $Y[0]*x^{64}+Y[2:1]$

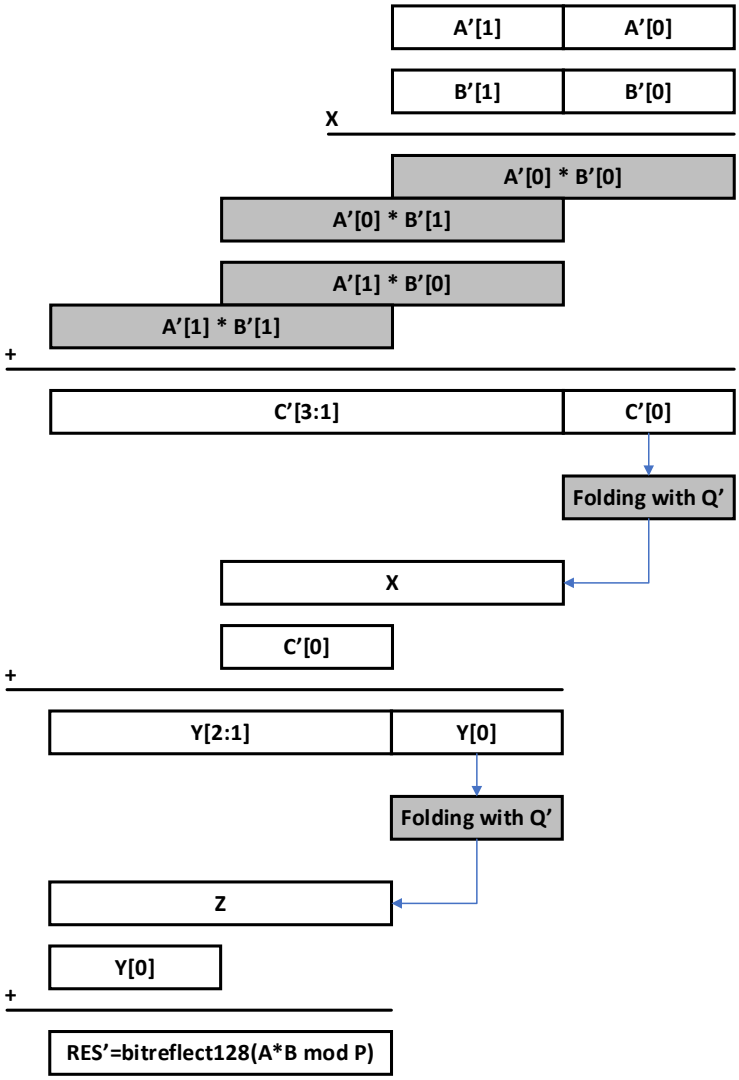

Figure 5. Software Friendly Multiply-Reduce Routine on Bit-Reflected Operands with Folding Approach

We can further optimize this algorithm with our pre-computation method. Optimized algorithm working on bit-reflected data is detailed as follows:

Table 10. Optimized Bit Reflected GHASH Algorithm

Inputs:	A'=bitreflect$_{128}$(A) B'=bitreflect$_{128}$(B) Q'=bitreflect$_{64}$(Q>>1) (Q=x^{128} mod P) K'=bitreflect$_{128}$(K)
Output:	RES'=A'*B' mod P'
Step 1:	**Compute Y** Y=GFMUL64(A'[0],K'[0])+ GFMUL64(A'[0],K'[1])*x^{64}+ GFMUL64(A'[1],B'[0])+ GFMUL64(A'[1],B'[1])*x^{64}
Step 2:	**Compute RES=C mod P** **Folding:** RES'=GFMUL64(Y[0],Q')+ Y[0]*x^{64}+Y[2:1]

This optimized bit-reflected algorithm is depicted in Figure 6.

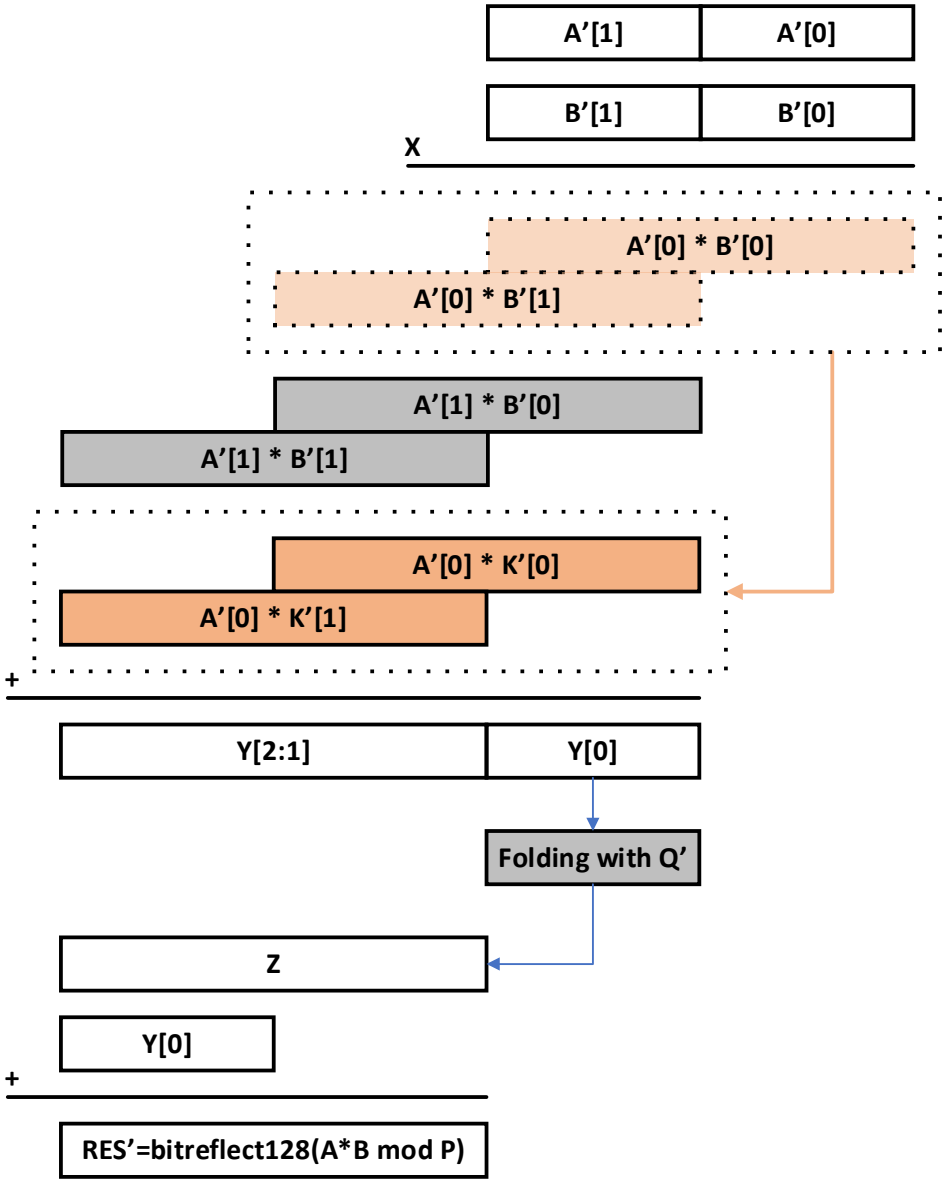

Figure 6. Software Friendly Multiply-Reduce Optimized Routine on Bit-Reflected Operands with Folding Approach

3 Parallel GHASH

We propose another novel method to further improve the performance of GHASH implementations. The GHASH algorithm is defined in *The Galois/Counter Mode of Operation (GCM)*[1]. The following operation constitutes the main building block of GHASH:

$$X_i = (X_{i-1} \oplus C_i) * H$$

where multiplication in GF(2^{128}) is denoted as *. As an example, assume GCM is defined over a packet with 0 Bytes of Additional Authentication Data (AAD) (m=0) and 12x16 Bytes of ciphertext (n=12). GHASH is defined as follows:

$$X_i = \begin{cases} (X_{i-1} \oplus C_i) * H & i = 1,2,\dots,12 \\ (X_{12} \oplus (len(A)||len(C))) * H & i = 13 \end{cases}$$

This is depicted in Figure 7.

[1] https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

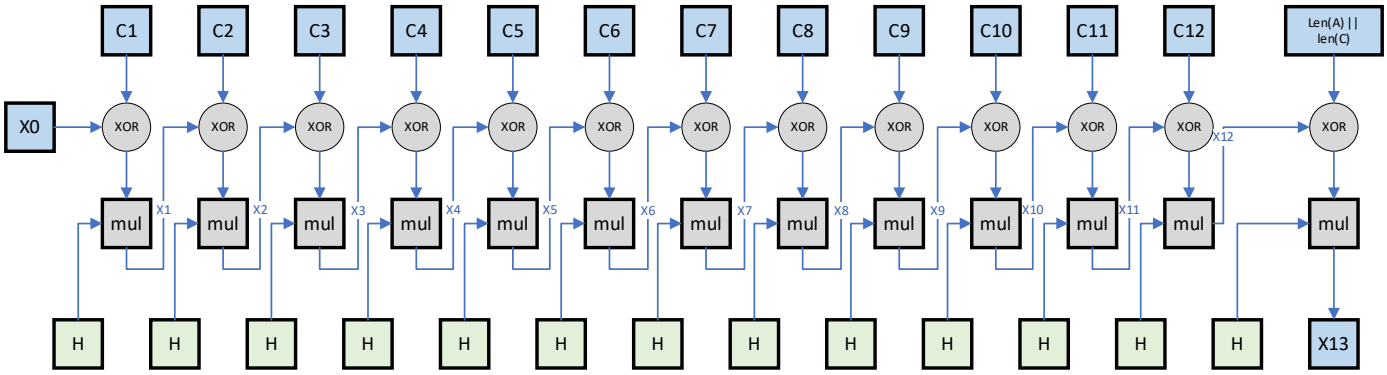

Figure 7. Example GHASH Computation on 12 Blocks of Ciphertext

The equation for X12 result can be written as:

$$X_{12} = X_0 * H^{12} + \sum_{i=1}^{12} C_i * H^{13-i}$$

GHASH is a serial operation. It is not suitable for software implementations. There are well-known methods to parallelize GHASH computation. Using Horner's method, and some precomputed values, the GHASH computation can be optimized as follows:

$$X_{12} = X_0 * H^{12} + C_1 * H^{12} + C_2 * H^{11} + C_3 * H^{10} + C_4 * H^9 + C_5 * H^8 + C_6 * H^7 + C_7 * H^6 + C_8 * H^5 + C_9 * H^4 + C_{10} * H^3 + C_{11} * H^2 + C_{12} * H^1$$

$$= ((((X_0 + C_1 * H^4) + C_2 * H^3 + C_3 * H^2 + C_4 * H^1) + C_5) * H^4 + C_6 * H^3 + C_7 * H^2 + C_8 * H^1 + C_9) * H^4 + C_{10} * H^3 + C_{11} * H^2 + C_{12} * H^1$$

The H^2, H^3 and H^4 values can be precomputed and used in the GHASH computation. Note that this is only an example. More parallelization can be achieved with more precomputed data over real workloads.

Four multiplication operations can run in parallel. A ZMM register can hold four 128-bit blocks, and with a single execution, four parallel GHASH operations can be realized.

In a SIMD setting, four multiplications can be parallelized very efficiently. However, after each of these four multiplications, there is an XOR step. All 128-bit sections of the ZMM registers need to be XORed together(see Figure 8).

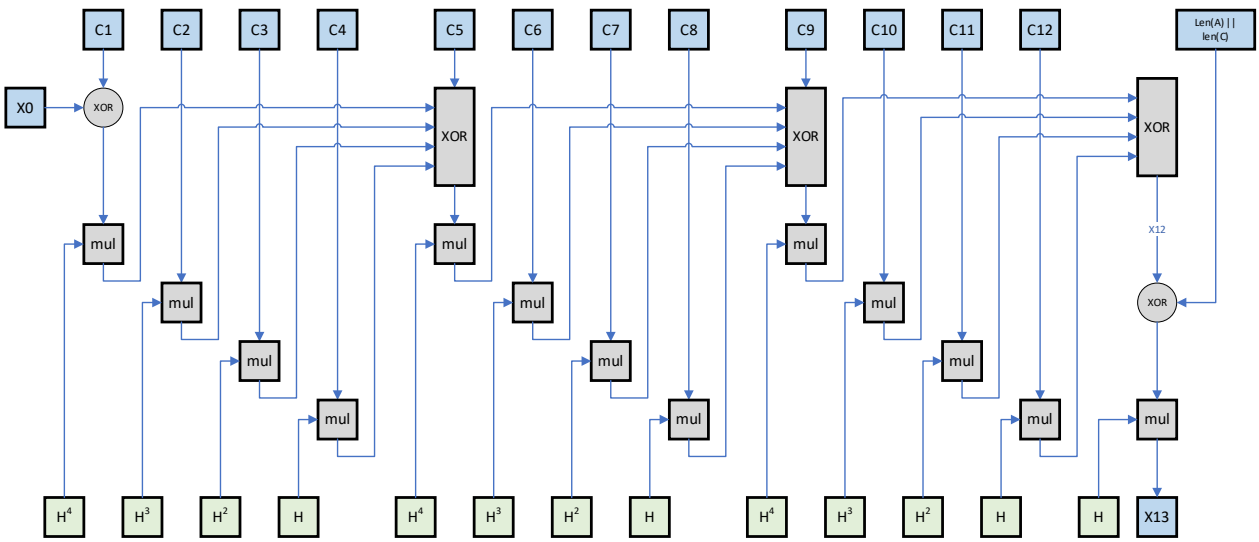

Figure 8. Parallelized GHASH computation

This can be eliminated with the following optimization (see Figure 9):

$$X_{12} =$$
$$\left(((X_0 + C_1) * H^4 + C_5) * H^4 + C_9\right) * H^4 +$$
$$\left((C_2 * H^4 + C_6) * H^4 + C_{10}\right) * H^3 +$$
$$\left((C_3 * H^4 + C_7) * H^4 + C_{11}\right) * H^2 +$$
$$\left((C_4 * H^4 + C_8) * H^4 + C_{12}\right) * H^1$$

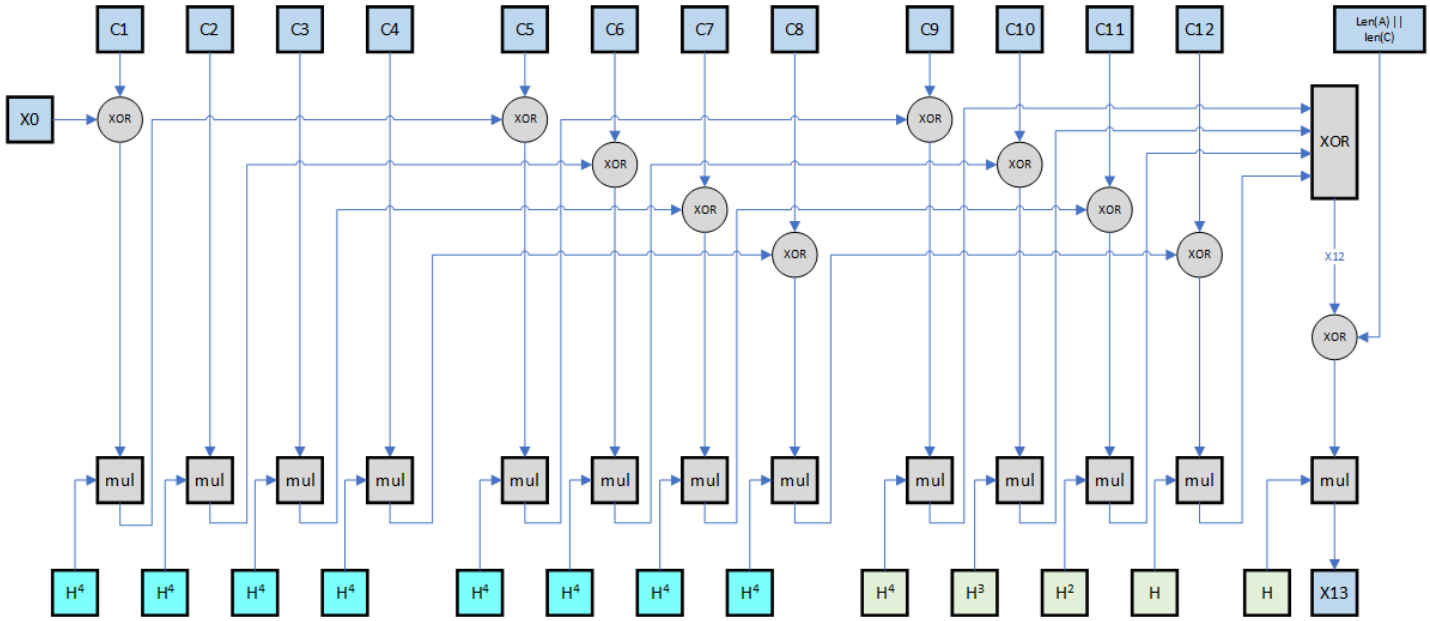

Figure 9. Further Optimized Parallel GHASH Computation

4 GHASH GF(2^{128}) Multiplication Example Code

Table 3 shows an example source code for a single block GHASH multiply and reduce operations. One column includes the baseline version without the optimization. The second column shows the new version code with the optimization. The schoolbook multiply part is identical in both cases but all steps that occur afterwards are hugely simplified in the optimized case. The new version needs only ten instructions to produce the result while the baseline version needs 18 instructions.

Table 11. Example code for GHASH multiply and reduce

Baseline version	New version
Input: xmm0 - output result / input block xmm1 - Q'	Input: xmm0 - output result / input block xmm1 - Q' xmm7 – K'
Output: xmm0	Output: xmm0
Instruction summary: - 7 x vpclmulqdq - 6 x vpxor - 5 x vpslldq / vpsrldq	Instruction summary: - 5 x vpclmulqdq - 4 x vpxor - 1 x vpshufd

`vpclmulqdq xmm2, xmm0, xmm1, 0x11`
`vpclmulqdq xmm3, xmm0, xmm1, 0x00`
`vpclmulqdq xmm4, xmm0, xmm1, 0x01`
`vpclmulqdq xmm0, xmm0, xmm1, 0x10`
`vpxor xmm0, xmm0, xmm4`
`vpsrldq xmm4, xmm0, 0x8`
`vpslldq xmm0, xmm0, 0x8`
`vpxor xmm2, xmm2, xmm4`
`vpxor xmm0, xmm0, xmm3`
`vmovdqa xmm4, [rel P]`
`vpclmulqdq xmm3, xmm4, xmm0, 0x01`
	`vpclmulqdq xmm5, xmm0, xmm7, 0x0`
	`vpclmulqdq xmm4, xmm0, xmm7, 0x10`
	`vpclmulqdq xmm3, xmm0, xmm1, 0x1`
	`vpclmulqdq xmm2, xmm0, xmm1, 0x11`
	`vpxor xmm5, xmm5, xmm3`
	`vpxor xmm2, xmm2, xmm4`
	`vpclmulqdq xmm0, xmm5, [rel P], 0x10`
	`vpshufd xmm3, xmm5, 0x4e`
	`vpxor xmm0, xmm0, xmm2`
	`vpxor xmm0, xmm0, xmm3`

vpslldq xmm3, xmm3, 0x8 vpxor xmm0, xmm0, xmm3 vpclmulqdq xmm3, xmm4, xmm0, 0x00 vpsrldq xmm3, xmm3, 0x4 vpclmulqdq xmm0, xmm4, xmm0, 0x10 vpslldq xmm0, xmm0, 0x4 vpxor xmm0, xmm0, xmm3 vpxor xmm0, xmm0, xmm2	

5 Performance Results

The performance measurement was done for encrypt direction and message sizes of 16384, 2048, 512, 256, 128 and 64 bytes. All measurements also included 12 bytes of Additional Authenticated Data (AAD), which is common for IPsec and TLS cases.

The comparison was done on Linux systems using Intel® Multi-Buffer Crypto for IPsec Library[2] and its performance tool. Two software versions have been tested:

- baseline - v1.3 version (`git checkout v1.3`)
- new - commit abd348a (`git checkout abd348aa85c29ed9b25bc0835897ff9ccf2838b3`)

"SAFE_OPTIONS=n" compilation option was used: `make -j SAFE_OPTIONS=n`

Example command line for AES-GCM-128 and AES-GCM-256 16384 bytes message benchmark:

- baseline version (v1.3)

```
env LD_LIBRARY_PATH=$PWD/lib ./perf/ipsec_perf --arch avx512 --aead-algo aes-gcm-128 --job-size 16384 --job-iter 500000 --no-time-box --cipher-dir encrypt

env LD_LIBRARY_PATH=$PWD/lib ./perf/ipsec_perf --arch avx512 --aead-algo aes-gcm-256 --job-size 16384 --job-iter 500000 --no-time-box --cipher-dir encrypt
```

- new version (commit abd348a)

```
env LD_LIBRARY_PATH=$PWD/lib ./perf/imb-perf --arch avx512 --aead-algo aes-gcm-128 --job-size 16384 --job-iter 500000 --no-time-box --cipher-dir encrypt

env LD_LIBRARY_PATH=$PWD/lib ./perf/imb-perf --arch avx512 --aead-algo aes-gcm-256 --job-size 16384 --job-iter 500000 --no-time-box --cipher-dir encrypt
```

Note that the default size in the performance tool is the 12 bytes AAD size selection.

Table 4 and Table 5 present performance improvement ratio between baseline implementation of AES-GCM for the 3rd and 4th Gen Intel® Xeon® Scalable processors.

The improvement ratio above value 1.00 means improvement of the new version vs the baseline, value below 1.00 would indicate that the new version is less efficient than the baseline one.

Improvement ratio = baseline [cycles] / new [cycles]

Table 12. AES-GCM-128 Improvement Ratios

Processor	Message Size [bytes]					
	16384	2048	512	256	128	64
3rd Gen Intel® Xeon ® Scalable Processor	1.09	1.05	1.16	1.14	1.12	1.26
4th Gen Intel® Xeon ® Scalable Processor	1.10	1.07	1.08	1.24	1.22	1.43

[2] https://github.com/intel/intel-ipsec-mb

Table 13. AES-GCM-256 Improvement Ratios

Processor	Message Size [bytes]					
	16384	2048	512	256	128	64
3rd Gen Intel® Xeon ® Scalable Processor	1.07	1.11	1.16	1.16	1.22	1.16
4th Gen Intel® Xeon ® Scalable Processor	1.00	1.02	1.09	1.26	1.23	1.38

The new version proves to be better than the baseline across all tested message sizes. For both, AES-GCM-128 and AES-GCM-256, improvement is most visible for small message sizes where reduction is performed three times in short succession one after another (GHASH of AAD, GHASH of the cipher text and GHASH of extra block with lengths). The improvement ratio decreases as the message size increases.

New AES-GCM implementation on 4th Gen Intel® Xeon® Scalable processor benefits higher performance gains than on its predecessor, the 3rd Gen Intel® Xeon® Scalable processor.

6 Benefits

We introduce two independent techniques to optimize the GHASH operation. The multiply-reduce optimization reduces the number of processor instructions required for the reduction operation without adding any complexity to the multiply operation. It benefits from the fact that one of the inputs to the multiply operation is always a precomputed value. It is a generic optimization that can be applied to any GHASH implementation. Parallel GHASH optimization also reduces the number of processor instructions, but only for a specific implementations case. SIMD implementations of GHASH benefit from this optimization technique.

Combined together, these optimizations drive significant improvement in efficiency of AES-GCM implementations on the 4th and 3rd Gen Intel® Xeon® Scalable processors, particularly for smaller packet sizes.

Another minor benefit of the optimized GHASH is that, in practical implementation, it frees a number of SIMD registers comparing to the baseline implementation. These registers can be used for some other data needed for the implementation.

It is worth noting that the techniques described here to optimize the GHASH function can be extended to other Galois Field multiplication applications and can be generalized for all GHASH implementations (hardware and software).

7 Summary

The optimization drives better performance and efficiency of AES-GCM cipher suite, which is the industry leading cipher suite used in secure network transport solutions like TLS, VPN or QUIC. The 2021 TLS Telemetry Report from F5 Lab analyzed top million sites and AES-GCM constituted 94% of the selected cipher suites in the top million sites.

This optimization improves performance and efficiency of this important cipher suite significantly on Intel® processors, especially for the harder-to-optimize case of small message sizes.

Appendix A System Configurations

Table 14. Software Configuration

	3rd Gen Intel® Xeon® Scalable Processor	4th Gen Intel® Xeon® Scalable Processor
Time	Thu May 18 01:52:15 PM UTC 2023	Thu 18 May 2023 12:23:02 PM UTC
Manufacturer	Inspur	Intel
CPU Model	Intel® Xeon® Gold 6348 CPU @ 2.60GHz	Intel® Xeon® Gold 6454S
Sockets	2	2
Cores per Socket	28	32
Hyperthreading	Enabled	Enabled
CPUs	112	128
Intel Turbo Boost	Disabled	Enabled *Note:* disabled in tests with "`echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo`" command
Base Frequency	2.6GHz	2.2GHz
All-core Maximum Frequency	3.4GHz	2.8GHz
Maximum Frequency	2.6GHz	3.4GHz
NUMA Nodes	4	2
Prefetchers	L2 HW, L2 Adj., DCU HW, DCU IP	L2 HW, L2 Adj., DCU HW, DCU IP
PPINs	831e2adc802b22bb,831c3ddc5650c09b	2294094a73af0a74,229408ce073cf107
Installed Memory	256GB (16x16GB DDR4 3200 MT/s [3200 MT/s])	16GB (1x16GB DDR5 4800 MT/s [4800 MT/s]); 32GB (1x32GB DDR5 4800 MT/s [4800 MT/s])
Hugepagesize	2048 kB	2048 kB
Transparent Huge Pages	madvise	madvise
Automatic NUMA Balancing	Enabled	Enabled
NIC	2x Ethernet Controller 10G X550T	1x Intel Corporation, 2x Ethernet Controller E810-C for QSFP
Disk	1x 447.1G INTEL SSDSCKKB48	1x 149.1G WDC_WD1600JS-00N
BIOS	05.01.01	EGSDCRB1.86B.0081.D18.2205301332
Microcode	0xd000389	0xaa000060
OS	Ubuntu 22.04.2 LTS	Ubuntu 20.04.5 LTS
Kernel	5.15.0-72-generic	5.14.0-051400-generic
TDP	235 watts	270 watts
Power & Perf Policy	Performance	Performance
Frequency Governor	powersave	powersave
Frequency Driver	intel_pstate	intel_pstate
Max C-State	9	9

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

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

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

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

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

0823/DN/WIT/PDF 783641-001US