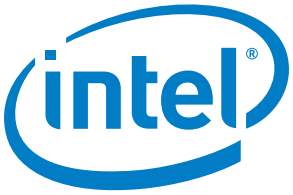


White Paper

Erdinc Ozturk
Vinodh Gopal

IA Architects
Intel Corporation

Enabling High-Performance Galois-Counter-Mode on Intel® Architecture Processors

October 2012

328154-001

Executive Summary

Galois-Counter Mode (GCM) is a block cipher mode of operation providing data security with AES encryption, and authentication with universal hashing over a binary field (GHASH). The main usage of GCM is in the IPSec, TLS 1.2 and SSH protocols – mostly for secure network communications.

With the recent introduction of AES-NI instructions (including PCLMULQDQ), highly-optimized implementations of GCM mode of operation were made possible on Intel® Architecture Processors. In this paper, we describe techniques to improve GCM performance further and describe a few versions of optimized code with performance data.

With our recent GCM implementations a single core of an Intel® Core™ i7 processor 2600 with Intel® HT Technology can compute GCM Encrypt on a large data buffer at the rate of ~2.2 cycles/byte[1]. Further performance improvements are expected in the 4th generation Intel® Architecture Processors. We expect and encourage the GCM mode to be more widely adopted due to performance improvements on Intel® Architecture Processors using the highly optimized code that we developed.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training,

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://www.intel.com/p/en_US/embedded.

Contents

Overview 5

Previous Work 5

Motivation 6

Implementation Details 6

Micro-architectural Optimizations 6

Further Parallelization 6

Optimized Multiplication 7

Optimized reduction 8

Critical-Path Optimization 9

Counter Increment and Shuffle operations 9

AES-CTR Mode Optimization for XOR operations 13

Software Versions 14

Performance 15

Methodology 15

Results 16

GCM Adoption 18

Conclusion 18

Acknowledgements 18

References 19

Overview

Galois-Counter Mode (GCM) is a block cipher mode of operation providing data security with AES encryption, and authentication with universal hashing over a binary field (GHASH) [2]. GCM is a component of Suite B Cryptography, which is a selection of cryptographic algorithms that has been approved by NSA for use by the U.S. Government and specified in NIST standards and recommendations.

GCM is a relatively new algorithm submitted to the National Institute of Standards and Technology (NIST) in January 2004. NIST published the approved GCM specifications in November 2007. The main usage of GCM is in the IPSec, 802.1AE MACSec, TLS 1.2 and SSH protocols – mostly for secure network communications.

With the recent introduction of AES-NI instructions (including PCLMULQDQ), highly-optimized implementations of GCM mode of operation were made possible on IA processors. In this paper, we describe recent techniques to improve GCM performance further and describe a few versions of optimized code with performance data.

We expect and encourage the GCM mode to be more widely adopted due to performance improvements on IA processors [11] using the highly optimized code that we developed.

Previous Work

In [4], we described the function stitching method which significantly improves performance of algorithm pairs running in parallel. Using the function stitching method, we improve the performance of GCM mode significantly as encryption and authentication algorithms are performed in parallel in this mode.

In [1], we described an optimized implementation of GCM combining function stitching with optimized polynomial large multiplication methods. Optimized code using those earlier techniques can be found in the Linux kernel (versions v2.6.38.3 onwards), in the aesni-intel_asm.s file under the /arch/x86/crypto folder [5]. System-level performance of IPSec using our earlier code was published in [3].

In this paper, we describe further optimizations for the GCM implementation via micro-architectural, algorithmic and implementation improvements.

Motivation

The primary motivations for the recent GCM code were to increase performance for recent and forthcoming IA processors, and to target usages other than IPSec. In particular, we see the opportunity of wider adoption of GCM in the SSL/TLS protocols, and developed optimized code for that usage as well.

In SSL/TLS, the records could be as large as 16KB in size. We also developed a new API of splitting the pre-computes from the main function, which enables the best performance when re-using keys across data buffers.

Implementation Details

Since optimized GCM code requires running two separate algorithms in parallel, the implementation of the algorithm is challenging as it utilizes a large portion of the architectural resources. Also, efficient implementations of GCM can reach very high instructions/cycle retirement rate, which makes every instruction critical for the performance of the algorithm. Even though some instructions can be taken outside of the critical path, because of micro-architectural limitations, the instruction might still be in the critical path of the execution pipeline of the CPU. We realized our optimizations considering these facts.

The optimizations can be summarized in 2 categories: Micro-architectural optimizations and Critical-path optimizations.

Micro-architectural Optimizations

AES-NI was first introduced in 2010 and it led to 3-10X improvements [7] in AES software implementations. As we showed in [1], efficient GCM implementations were also possible with AES-NI. GCM performance increases with improved throughput performance of AES-NI, by enabling further parallelization of the AES portion of GCM algorithm.

Further Parallelization

If the throughput of the AES-NI instructions is improved (including improvements in PCLMULQDQ instructions), increased parallelization of the functions becomes feasible. In our previous paper, we implemented GCM using a by-4 approach, which means that we applied both AES-CTR mode operations and GHASH on 4 blocks (or 64-byte chunks) at a time. With increased throughput of the AES-NI, further parallelization is possible and we could increase the chunk sizes to 128-bytes (a by-8 approach), resulting in a significant increase in performance.

Optimized Multiplication

In our previous paper, we implemented the 128-bit multiplication components of the GHASH algorithm using the Karatsuba multiplication algorithm[6]. With a classical approach, 4 PCLMULQDQ instructions are required for a 128-bit multiplication. With the Karatsuba Algorithm, this number drops to 3. Even though the Karatsuba approach decreases the number of core multiplications, it introduces overhead. For those micro-architectures where the PCLMULQDQ performance has improved [11] to have better performance than the Karatsuba overhead, we can implement the 128-bit multiplications using the classical approach. The difference is shown in Table 1.

Table 1. Classical Multiplication vs. Karatsuba Multiplication

Classical Approach	Karatsuba Approach
	movdqa T1, GH pshufd T2, GH, 01001110b pshufd T3, HK, 01001110b ; T2 = (a1+a0) pxor T2, GH ; T3 = (b1+b0) pxor T3, HK
movdqa T1, GH *; T1 = a1*b1* **pclmulqdq** T1, HK, 0x11 movdqa T2, GH *; T2 = a0*b0* **pclmulqdq** T2, HK, 0x00 movdqa T3, GH *; T3 = a1*b0* **pclmulqdq** T3, HK, 0x01 *; GH = a0*b1* **pclmulqdq** GH, HK, 0x10	; T1 = a1*b1 **pclmulqdq** T1, HK, 0x11 ; GH = a0*b0 **pclmulqdq** GH, HK, 0x00 ; T2 = (a1+a0)*(b1+b0) **pclmulqdq** T2, T3, 0x00
pxor GH, T3 movdqa T3, GH psrldq T3, 8 pslldq GH, 8	pxor T2, GH pxor T2, T1 movdqa T3, T2 pslldq T3, 8 psrldq T2, 8
; *<T1:GH> holds the result of the carry-less multiplication of GH by HK*	*; <T1:GH> holds the result of the carry-less multiplication of GH by HK*

Classical Approach	Karatsuba Approach
pxor T1, T3 pxor GH, T2	pxor GH, T3 pxor T1, T2

Optimized reduction

The GHASH algorithm requires modular multiplication of 128-bit numbers and since the algorithm utilizes a simple modulus polynomial, a shift-based reduction [9] can be utilized as we did in [1]. An alternative approach uses the PCLMULQDQ instruction for reduction. As stated before, the performance depends on the throughput of the PCLMULQDQ instruction. The different algorithms are shown in Table 2.

Table 2. Shift-based Reduction vs. PCLMULQDQ-based reduction

Shift-based reduction[9]	PCLMULQDQ-based reduction
;first phase of the reduction movdqa T2, GH movdqa T3, GH movdqa T4, GH pslld T2, 31 pslld T3, 30 pslld T4, 25 pxor T2, T3 pxor T2, T4 movdqa T5, T2 psrldq T5, 4 pslldq T2, 12 pxor GH, T2 ;second phase of the reduction movdqa T2,GH movdqa T3,GH movdqa T4,GH psrld T2,1 psrld T3,2 psrld T4,7 pxor T2,T3 pxor T2,T4 pxor T2, T5 pxor GH, T2	;first phase of the reduction movdqa %%T3, [POLY2 wrt rip] movdqa %%T2, %%T3 pclmulqdq %%T2, %%GH, 0x01 pslldq %%T2, 8 pxor %%GH, %%T2 ;second phase of the reduction movdqa %%T2, %%T3 pclmulqdq %%T2, %%GH, 0x00 psrldq %%T2, 4 pclmulqdq %%GH, %%T3, 0x01 pslldq %%GH, 4 pxor %%GH, %%T2 pxor %%GH, %%T1

Shift-based reduction[9]	PCLMULQDQ-based reduction
pxor GH, T1	

Critical-Path Optimization

The mode of operation of AES in the GCM mode is the counter or CTR mode. We describe optimizations that improve performance of counter mode processing by alleviating the critical-path.

Counter Increment and Shuffle operations

The plaintext input to the AES algorithm has to be byte-reflected due to AES specifications. For a buffer size of n 16-Byte blocks, the algorithm for encryption portion of GCM is:

```
Initialize CTR as Y0 (as explained in [2])
for (i = 0 ; i < n ; i++)
{
       CTR = CTR+1
       xmm1   = byte_reflect(CTR) //realized with a pshufb instruction
       xmm1   = AES(xmm1, Key)
       ciphertext = xmm1 XOR plaintext
}
```

This algorithm is illustrated in Figure 1.

We devised an algorithm that eliminates the need for a pshufb instruction. We implement the increment of the counter value by adding a 1 to the most significant byte of this value. The pseudocode is:

```
for (i = 0 ; i < n ; i++)
{
    CTR = CTR + (1 << (128-8))
    xmm1   = AES(CTR, Key)
    ciphertext = xmm1 XOR plaintext
}
```

This algorithm is illustrated in Figure 2.

However, we need to implement careful looping to ensure carry propagations are handled correctly in the 32-bit counter field, which results in the following:

```
index = msb(CTR)
for (i = 0 ; i < n ; i++)
{
    if (index == 0xFF)
        CTR = byte_reflect(CTR)
        CTR = CTR+1
```

```
            xmm1   = byte_reflect(CTR) //realized with a pshufb instruction
            xmm1   = AES(xmm1, Key)
            ciphertext = xmm1 XOR plaintext
            index = 0
        else
            CTR = CTR + (1 << (128-8))
            xmm1   = AES(CTR, Key)
            ciphertext = xmm1 XOR plaintext
            index  = (index + 1) mod (0x100)
    }
```

This algorithm is illustrated in Figure 3. Note that the fast-path with no shuffle/byte-reflections is executed the majority of the time, yielding a performance gain.

Figure 1. AES-CTR mode operations of GCM

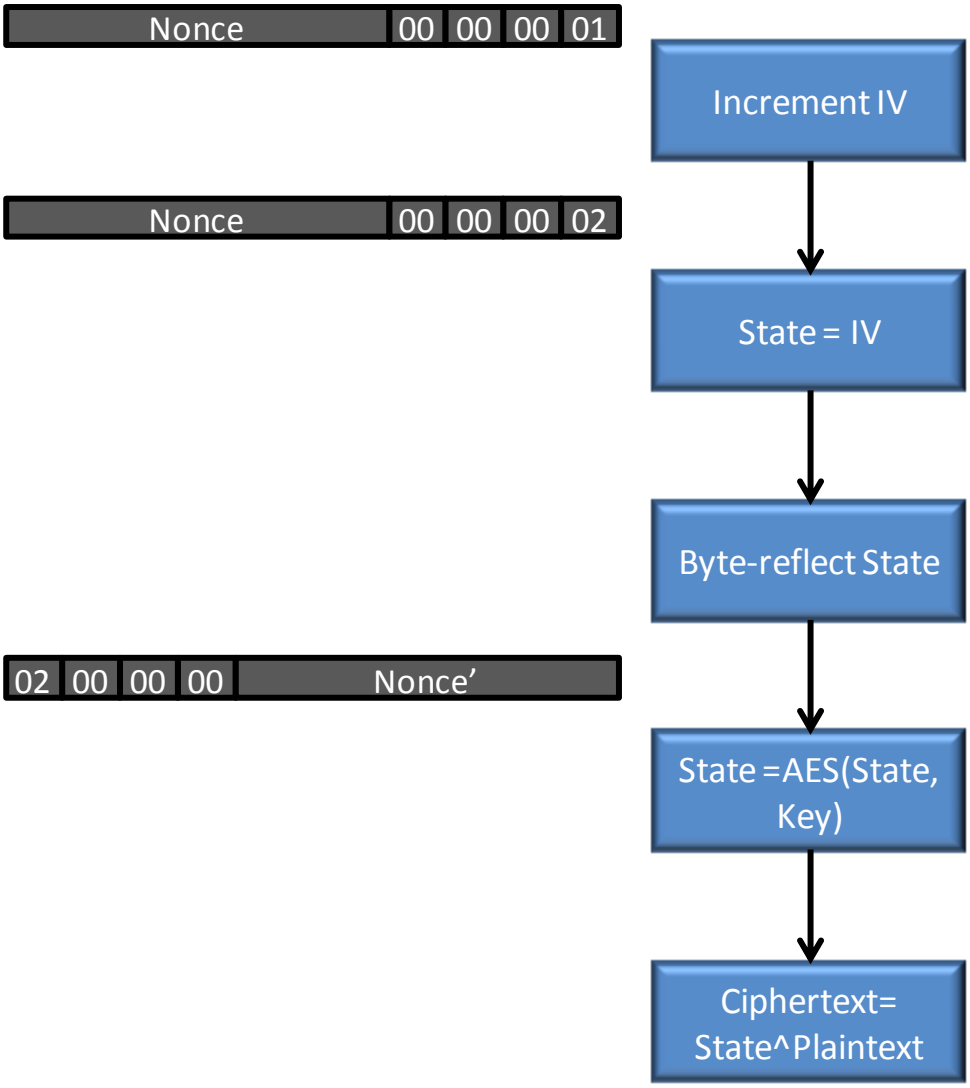


Figure 2. Optimized main loop of AES-CTR mode of GCM

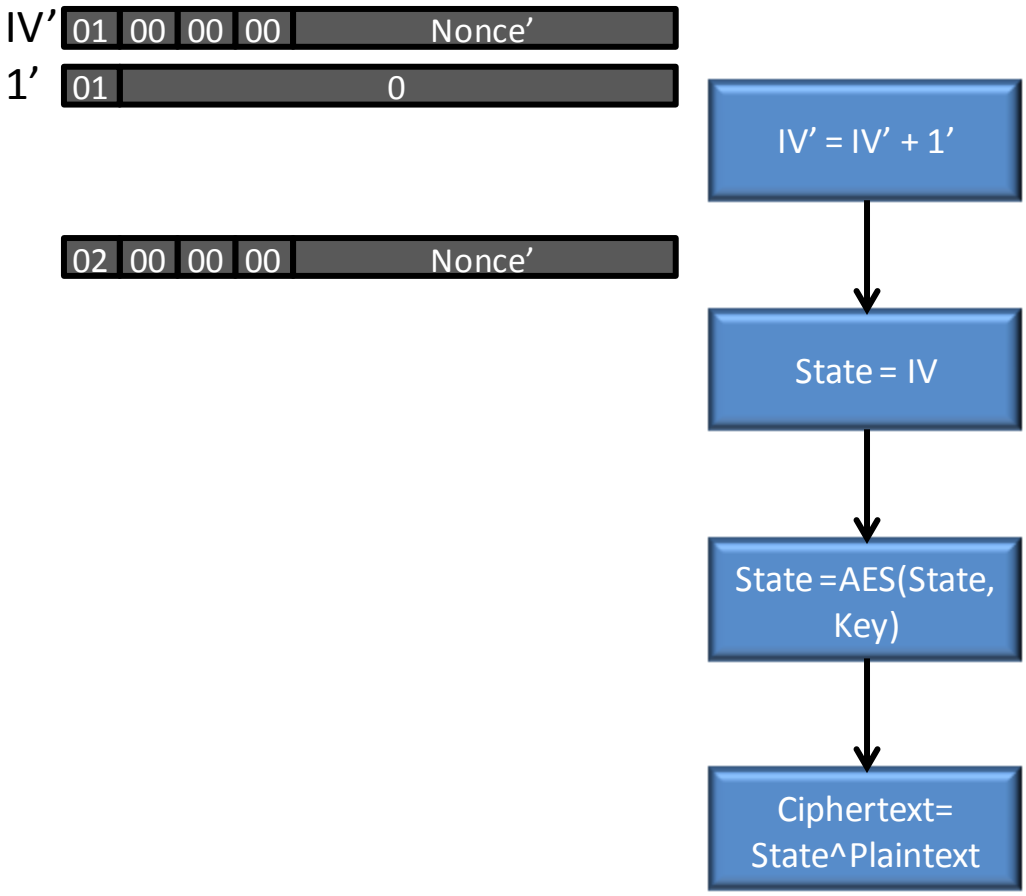
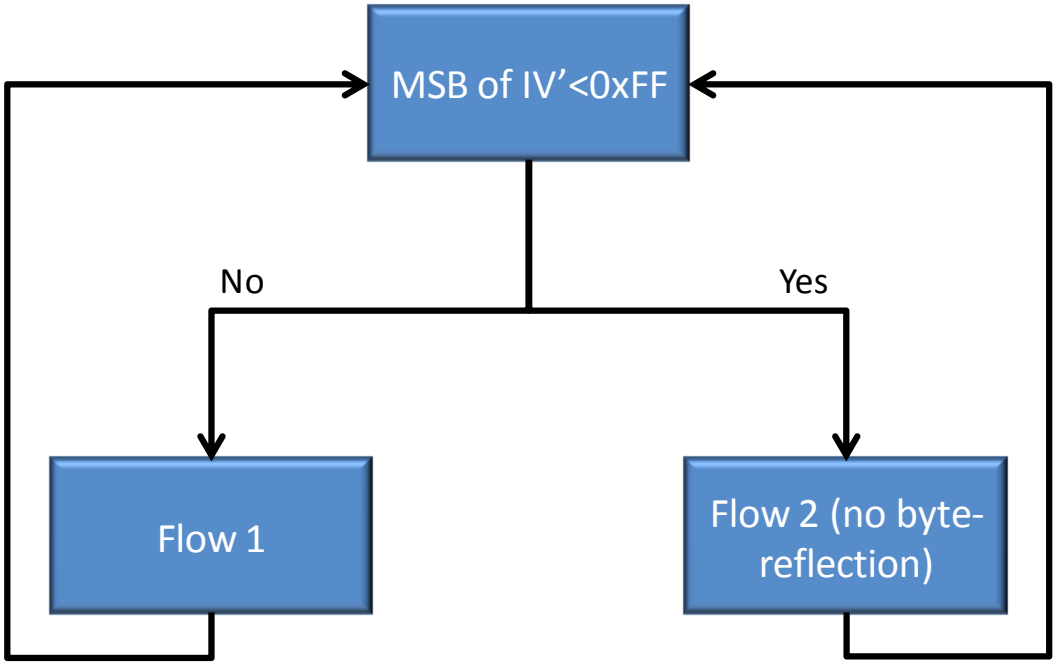


Figure 3. Optimized loop flow



AES-CTR Mode Optimization for XOR operations

The AES-CTR mode algorithm is implemented as follows:

```
xmm0 = CTR + 1
aesenc xmm0, key0
…
aesenc xmm0, key9
aesenclast xmm0, key10
pxor xmm0, ptext
```

Here the last pxor operation is in the critical path of the operation as it has to wait for the result of the aesenclast instruction. Aesenclast instruction is defined by the pseudocode in [7]:

```
AESENCLAST xmm1, xmm2/m128
    Tmp := xmm1;
    Round Key := xmm2/m128;
    Tmp := Shift Rows (Tmp);
    Tmp := SubBytes (Tmp);
    xmm1 := Tmp xor Round Key
```

As can be seen, the last operation in the aesenclast instruction is XOR with the round key, which is key10 in our implementation. This means that we can XOR the ptext with key10 before the aesenclast instruction, which will remove the xor operation from the critical path. The new pseudocode is:

```
xmm0 = CTR + 1
aesenc xmm0, key0
…
aesenc xmm0, key9
aesenclast xmm0, (key10^ptext)
```

Similar techniques to improve performance of some modes of AES have been described in [8].

Software Versions

In [1], we explained the methodology for implementing highly optimized GCM software by pre-computing certain values related to the Hashkey. To efficiently parallelize (by-8) GHASH computations, the following values are precomputed using Hashkey H:

H^2, H^3, H^4, H^5, H^6, H^7, H^8

Since Hashkey H is directly related to the AES key used for GCM and the same key can be used for multiple GCM computations in some application contexts, we separated the pre-compute operations as a separate function. This resulted in 3 functions: precompute, encrypt and decrypt.

For each of the functions mentioned above, three versions of software are available in [6], which we refer to as:

1. gcm_sse.asm: This version uses the SSE instruction set and is optimized for 1st generation Core architecture. The Karatsuba Algorithm is used for multiplication and shift-based reduction is used for the 128-bit reduction.
2. gcm_gen2.asm: This version uses the AVX1 instructions and is optimized for 2nd generation Core architecture. Algorithms used for the implementation are identical to the SSE version.
3. gcm_gen4.asm: This version also uses the AVX1 instruction set. However, Classical Multiplication Algorithm is used for multiplication and multiplication-based reduction is used for the 128-bit reduction.

The applications that can re-use keys over many data buffers should call the precompute function once to generate the expanded AES key schedule and the GHASH precomputed values. The data buffers are then processed using either the encrypt or decrypt functions that also take in the precomputed values as additional input.

Applications that cannot re-use keys across many data buffers will need to call the precompute function once for every buffer encrypted or decrypted. However for such usages, once could create a combined function that includes precomputation with slightly better performance, utilizing the stitching method to parallelize precompute and main functions as much as

possible. A combined function would also be beneficial when there is no key re-use across small data buffers, because in those cases, we could selectively compute as many precompute values as needed; thus, based on the length of the data buffer, the function could determine whether a by-4 or a by-8 method is appropriate and then compute only the required constants.

Note that the gcm_gen4 code has been optimized to run on the 4th generation processors and beyond, and should not be used on the previous generations (such as the 2nd generation), as there are better performing versions designed for the previous generations.

For the IPSec usage, it would be better to upgrade from our earlier code in [5] to our new by-8 SSE code, especially if the application context is being modified to cache key material and re-use them across data buffers.

Performance

The performance results provided in this section were measured on widely available Intel® Processors. The SSE version was run on an Intel® Xeon® processor X5670 and Intel® Core™ i7 processor 2600, and the AVX1 versions were run on an Intel® Core™ i7 processor 2600. In each case the performance of 512, 1024, 2048, 4096 and 8192-Byte buffer sizes was measured. The tests were run with Intel® Turbo Boost Technology off.

Methodology

We measured the performance of the functions on data buffers of different sizes. We called the functions on the same buffer a large number of times, collecting many timing measurements. For each data buffer, we then sorted the timings, discarded the top and bottom 1/8th samples and then the largest/smallest quarter, and averaged the remaining quarter.

The timing was measured using the rdtsc() function which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The 'TSC_initial' is the TSC recorded before the function is called. After the function is complete, the rdtsc() was called again to record the new cycle count 'TSC_final'. The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial).

A large number of such measurements were made for each data buffer and then averaged as described above to get the number of cycles for that buffer size. Finally, that value was divided by the buffer size to express the performance in cycles per byte.

Results

We show performance in cycles/byte for varying sizes of input data buffers in Table 3. The code we developed has not been optimized for small data buffer performance.

Table 3. Performance data (cycles/byte) for GCM Encrypt and Decrypt functions[2]

SSE-ENCRYPT		512	1024	2048	4096	8192
X5670	single thread	3.37	3.14	3.03	2.99	2.97
	HT	3.04	2.88	2.80	2.77	2.75
i7 2600	single thread	3.02	2.84	2.75	2.71	2.69
	HT	2.59	2.45	2.38	2.34	2.33

SSE-DECRYPT		512	1024	2048	4096	8192
X5670	single thread	3.42	3.21	3.11	3.05	3.04
	HT	3.07	2.91	2.84	2.80	2.79
i7 2600	single thread	3.05	2.89	2.81	2.77	2.75
	HT	2.65	2.51	2.45	2.41	2.39

AVX-GEN2-ENCRYPT		512	1024	2048	4096	8192
i7 2600	single thread	2.96	2.78	2.71	2.65	2.68
	HT	2.52	2.37	2.30	2.26	2.24

[2] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

AVX-GEN2-DECRYPT		512	1024	2048	4096	8192
i7 2600	single thread	2.94	2.74	2.67	2.61	2.59
	HT	2.52	2.36	2.30	2.25	2.23

AVX-GEN4-ENCRYPT		512	1024	2048	4096	8192
i7 2600	single thread	3.67	3.48	3.39	3.34	3.32
	HT	3.06	2.91	2.84	2.80	2.78

AVX-GEN4-DECRYPT		512	1024	2048	4096	8192
i7 2600	single thread	3.70	3.48	3.38	3.34	3.31
	HT	3.06	2.90	2.83	2.79	2.77

It can be seen that HT brings ~10% speedup on X5670 and ~20% on i7 2600. Decrypt is slightly lower performance than encrypt by ~2%. The best performance can be seen for large buffers on the i7 2600, using the AVX-GEN2 Encrypt code – a single core can achieve a performance of 2.24 Cycles/Byte.[3]

[3] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

GCM Adoption

GCM adoption has not been very fast to-date, due to its recent standardization and the fact that Cipher Suites were added to **TLS 1.2** very recently (August 2008). Some facts to note:

- Only one web browser today supports GCM (IE 9 on Windows 7, where GCM is not default and is the 10th choice on the cipher-suite list).
- iOS does not support GCM, despite supporting TLS 1.2 by default.

NSS does not support TLS 1.2, which impacts the Chrome and Firefox browsers. TLS 1.2 is not widely used today, but we are hoping to see increased adoption with OpenSSL. Given the performance on IA Processors, we believe the ecosystem should increase the ramp of GCM for authenticated encryption.

A recent draft standard in [10] proposes an update to the Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH) for the IPSec protocol, and adds usage guidance to help in the selection of these algorithms. Notably it emphasizes and elevates the importance of AES-GCM in IPSec.

Conclusion

In this paper we described three highly-optimized GCM implementations optimized for Intel® Architecture processors. We described various algorithm and implementation optimizations we used in this GCM code, and presented performance data. On the 2nd generation Intel® Core™ i7 processor 2600, a single core can achieve GCM Encrypt performance of 2.24 Cycles/Byte on large data buffers.[4]

Acknowledgements

We thank Wajdi Feghali, Jim Guilford, Gilbert Wolrich and Sean Gulley for their substantial contributions.

[4] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

References

[1] Vinodh Gopal, Erdinc Ozturk, Wajdi Feghali, Jim Guilford, Gil Wolrich, Martin Dixon. Optimized Galois-Counter-Mode Implementation on Intel® Architecture Processors. Intel White Paper, August 2010.

[2] David A. McGrew, John Viega. The Galois/Counter Mode of Operation (GCM). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf.

[3] Adrian Hoban. Using Intel® AES New Instructions and PCLMULQDQ to Significantly Improve IPSec Performance on Linux. Intel White Paper, August 2010.

[4] Vinodh Gopal, Wajdi Feghali, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Martin Dixon, Max Locktyukhin, Maxim Perminov. Fast Cryptographic Computation on IA processors via Function Stitching. Intel White Paper, April, 2010.

[5] GCM Linux patch http://lkml.indiana.edu/hypermail/linux/kernel/1010.1/01087.html

[6] Optimized AES GCM Software http://www.intel.com/p/en_US/embedded/hwsw/software/crc-license?id=6386&iid=6387

[7] Intel® Advanced Encryption Standard (AES) New Instructions Set http://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf

[8] Manley, R., Gregg, D. "A program generator for Intel AES-NI instructions." in: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 311–327. Springer (2010)

[9] Intel Corporation: Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode — Rev 2 (2010)

[10] D McGrew, W Feghali "Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH)" http://www.ietf.org/internet-drafts/draft-mcgrew-ipsec-me-esp-ah-reqts-00.txt

[11] Intel's Haswell Architecture Analyzed http://www.anandtech.com/show/6355/intels-haswell-architecture/8

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://www.intel.com/p/en_US/embedded.

Authors

Erdinc Ozturk and **Vinodh Gopal** are IA Architects with the Intel Architecture Group at Intel Corporation.

Acronyms

IA Intel® Architecture

§