Making AES great again: the forthcoming vectorized AES instruction

Nir Drucker[1,2] and Shay Gueron[1,2]

[1]University of Haifa, Haifa, Israel,
and
[2]Amazon Web Services Inc., Seattle, WA , USA*

Abstract. The introduction of the processor instructions AES-NI and `VPCLMULQDQ`, that are designed for speeding up encryption, and their continual performance improvements through processor generations, has significantly reduced the costs of encryption overheads. More and more applications and platforms encrypt all of their data and traffic. As an example, we note the world wide proliferation of the use of AES-GCM, with performance dropping down to 0.64 cycles per byte (from ~ 23 before the instructions), on the latest Intel processors. This is close to the theoretically achievable performance with the existing hardware support. Anticipating future applications and increasing demand for high performance encryption, Intel has recently announced [1] that its future architecture (codename "Ice Lake") will introduce new encryption instructions. These will be able to vectorize the AES-NI and `VPCLMULQDQ` instructions, on wide registers that are available on the AVX512 architectures. In this paper, we explain how these new instructions can be used effectively, and how properly using them can lead to the anticipated theoretical encryption throughput of around 0.16 cycles per byte. The included examples demonstrate AES encryption in various modes of operation, AEAD such as AES-GCM, and the emerging nonce misuse resistant variant AES-GCM-SIV.

1 Introduction

AES is the most ubiquitous symmetric cipher, used in many applications and scenarios. A prominent example is the exponentially growing volume of encrypted online data. Evidence for this growth, which is strongly supported by the industry (e. g., Intel's new AES-NI instructions [2–4], and Google's announcement [5] on favoring sites that use HTTPS) can be observed, for example, in [6] showing that more than 70% of online websites today use encryption.

This makes the performance of AES a major target for optimization in software and hardware. Dedicated hardware solutions were presented (e. g., [7, 8]) and via the introduction of the AES-NI instructions that were added to x86 general purpose CPUs (and other architectures). These instructions, together with the progress made in processors' microarchitectures, allow software to run

* This work was done prior to joining Amazon.

the Authenticated Encryption with Additional Authentication Data (AEAD) scheme AES-GCM at 0.64 cycles per byte (C/B hereafter), approaching the theoretical performance of encryption only, 0.625 C/B, on such CPUs. Other software optimizations, written in OpenCL or CUDA that aim for the Graphical Processor Unit (GPU) [9,10] achieve the performance of 0.56 C/B and 0.44 C/B, respectively. Last year, AMD introduced the new "Zen" processor that has two AES units [11], and this reduces the theoretical throughput of AES encryption to 0.31 C/B.

Recently, Intel has announced [1] that its future architecture, microarchitecture codename "Ice Lake", will add vectorized capabilities to the existing AES-NI instructions, namely `VAESENC`, `VAESENCLAST`, `VAESDEC`, and `VAESDECLAST` (`VAES*` for short). These instructions are intended to push the performance of AES software further down, to a new theoretical throughput of 0.16 C/B.

This can directly speed up AES modes such as AES-CTR and AES-CBC, and also more elaborate schemes such as AES-GCM and AES-GCM-SIV [12,13] (a nonce misuse resistant AEAD). These two schemes require fast computations of the almost XOR-universal hash functions GHASH and POLYVAL, which are significantly sped up with dedicated "carry-less multiplication" instruction `PCLMULQDQ` [3,14]. Indeed, fast AES-GCM(-SIV) implementations can be achieved by using the new instruction that vectorizes the `PCLMULQDQ` instruction (`VPCLMULQDQ`) (see [15]).

In this paper, we demonstrate how to write software that efficiently uses the new `VAES*` and `VPCLMULQDQ` instructions. While the correctness of our algorithms (and code) can be verified with existing public tools, the actual performance measurements require a real CPU, which is currently unavailable. To address this difficulty, we give predictions based on instructions' count of current and new implementations.

The paper is organized as follows. Section 2 describes the new `VAES*` and `VPCLMULQDQ` instructions. Section 3 describes our implementations of AES encryption modes AES-CTR and AES-CBC. Section 4 focuses on the AEAD schemes AES-GCM and AES-GCM-SIV. In Section 5, we explain our results, and we conclude in Section 6.

2 Preliminaries

We use AES to refer to AES128. The xor operation is denoted by $\oplus$, and concatenation is denoted by $||$ (e. g., $00100111||10101100 = 0010011110101100$, which, in hexadecimal notation, is the same as `0x27` $||$ `0xac` = `0x27ac`). The notation $X[j:i]$, $j > i$ refers to the values of an array X between positions i and j (included). The case $i = j$ degenerates to $X[i]$. Here, X can be an array of bits or of bytes, depending on the context. For an array of bytes X, we denote by $\overline{X}$ the corresponding byte swapped array (e. g., $X =$`0x1234`, $\overline{X} =$`0x3412`). The two new vectorized AES-NI and `PCLMULQDQ` instructions are described next. The description of other assembly instructions can be found in [16].

2.1 Vectorized AES-NI

Intel's AES-NI instructions (AES*) include AESKEYGENASSIST and AESIMC to support AES key expansion and AESENC/DEC(LAST) to support the AES encryption/decryption, respectively. Alg. 1 illustrates the new VAES* instructions. These are able to perform one round of AES encryption/decryption on $KL = 1/2/4$ 128-bit operands (two qwords), having both register-memory and register-register variant (we use only the latter here). The inputs are two source operands, which are 128/256/512-bit registers (named xmm, ymm, zmm, respectively), that (presumably) represent the round key and the state (plaintext/ciphertext). The special case $KL = 1$ using xmm registers degenerates to the current version of AES*.

Algorithm 1 VAES*, and VPCLMULQDQ instructions [1]

```
Inputs: SRC1, SRC2 (wide registers)
Outputs: DST (a wide register)
procedure VAES*(SRC1, SRC2)
    for $i := 0$ to $KL - 1$ do
        $j = 128i$
        RoundKey[127 : 0] = SRC2[$j + 127 : j$]
        T[127 : 0] = (Inv)ShiftRows(SRC1[$j + 127 : j$])
        T[127 : 0] = (Inv)SubBytes(T[127 : 0])
        T[127 : 0] = (Inv)MixColumns(T[127 : 0])
                                        ▷ Only on VAESENC/VAESDEC.
        DST[$j + 127 : j$] = T[127 : 0] $\oplus$ RoundKey[127 : 0]
    return DST
```

```
Inputs: SRC1, SRC2 (wide registers) Imm8 (8 bits)
Outputs: DST (a wide register)
procedure VPCLMULQDQ(SRC1, SRC2, Imm8)
    for $i := 0$ to $KL - 1$ do
        $j_1 = 2i + Imm8[0]$
        $j_2 = 2i + Imm8[4]$
        T1[ 63 : 0 ] = SRC1[ $64(j_1 + 1) - 1 : 64j_1$ ]
        T2[ 63 : 0 ] = SRC2[ $64(j_2 + 1) - 1 : 64j_2$ ]
        DST[ $128(i + 1) - 1 : 128i$ ] = PCLMULQDQ( T1, T2 )
    return DST
```

2.2 Vectorized **VPCLMULQDQ**

Alg. 1 (bottom) illustrate the functionality of the new vectorized VPCLMULQDQ instruction. It vectorizes polynomial (carry-less) multiplication, and is able to perform $KL = 1/2/4$ multiplications of two qwords in parallel. The 64-bit multiplicands are selected from two source operands and are determined by the value of the immediate byte. The case $KL = 1$ degenerates to the current `VPCLMULQDQ` instruction.

3 Accelerating AES with VAES*

The use of the VAES* instructions for optimizing the various uses of AES is straightforward for some cases (e. g., AES-ECB, AES-CTR, AES-CBC decryption). For example, to optimize AES-CTR, which is a naturally parallelizable mode, we only need to replace each xmm with zmm register and handle the counter in a vectorized form. In some other case, using the new instruction is more elaborate (e. g., optimizing AES-CBC encryption, AES-GCM, or AES-GCM-SIV).

Fig. 1 compares legacy (Panel a.) and vectorized (Panel b.) codes of AES-CTR. In both cases, the counter is loaded and incremented first (Steps 6-8 and 7-8, respectively). In Panel b., Steps 9-11, the key schedule is duplicated 4 times in 11-zmm registers (zmm0-zmm10). The encryption is executed in Steps 9-13, and 12-16, of Panels a and b, respectively. Finally, the plaintext is xored and the results are stored.

(a) Legacy AES-CTR

```
 1 .set t, %xmm12
 2 .set ctrReg, %xmm11
 3 inc_mask:
 4 .long 0,0,0,0x01000000
 5
 6 vmovdqu (ctr), ctrReg
 7 .irp j,1,2,3,4
 8  vpadd inc_mask(%rip),ctrReg
 9  vpxor (key),ctrReg,t
10  .irp i,1,2,3,4,5,6,7,8,9
11   vaesenc \i*0x10(key),t,t
12  .endr
13  vaesenclast 10*0x10(key),t,t
14  vpxor (pt),t,t
15  vmovdqu tmp, (ct)
16  lea 0x10(pt), pt
17  lea 0x10(ct), ct
18 .endr
```

(b) Vectorized AES-CTR

```
 1 .set t, %zmm12
 2 .set ctrReg, %zmm11
 3 inc_mask:
 4 .long 0,0,0,0x01000000,0,0,0,0x02000000
 5 .long 0,0,0,0x03000000,0,0,0,0x04000000
 6
 7 vbroadcasti64x2 (ctr),ctrReg
 8 vpadd inc_mask(%rip),ctrReg
 9 .irp i,0,1,2,3,4,5,6,7,8,9,10,11
10  vbroadcasti64x2 \i*0x10(key),%zmm\i
11 .endr
12 vpxorq %zmm0, ctrReg, t
13 .irp i,1,2,3,4,5,6,7,8,9
14  vaesenc %zmm\i,t,t
15 .endr
16 vaesenclast %zmm10,t,t
17 vpxorq (pt),t,t
18 vmovdqu64 t,(ct)
```

Fig. 1. AES-CTR sample (AT&T assembly syntax); ct[511 : 0]=AES-CTR(pt[511 : 0], key).

A mode like AES-CBC encryption is serial by nature, and cannot be parallelized. However, we note that the VAES* instructions encrypt 2/4 independent plaintext streams in parallel. To do this, we need to rearrange ("transpose") the inputs/outputs in order to make them suitable for vectorized code such as in Fig. 1.

Fig. 2 illustrates how to handle four independent $4*128$-bit plaintext streams (A, B, C, D). We first load the four 512-bit values into four zmm registers (red

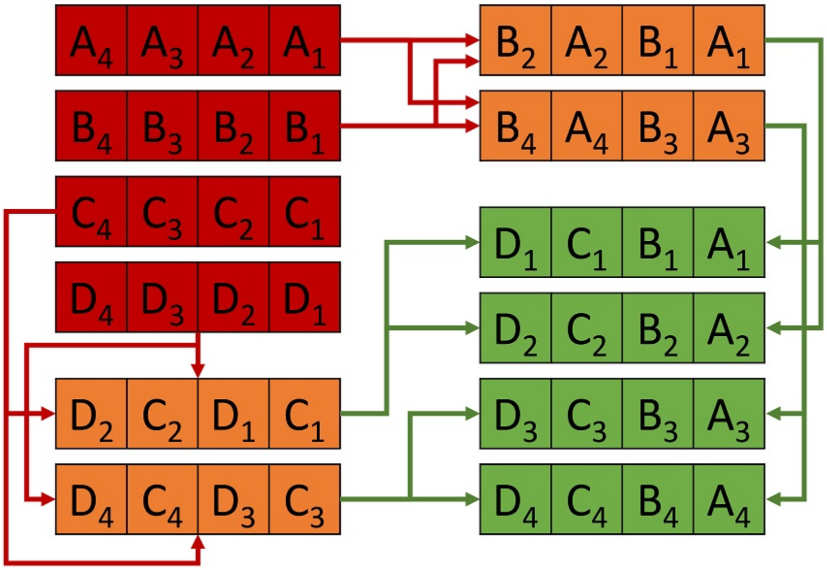

Fig. 2. Transposing a 4×4 128-bit (input in red; output in green) by executing eight VPERMI2Q instructions. Every group of four instructions can run in parallel.

upper left vectors), then use the VPERMI2Q instruction to permute the qwords of each two vectors A, B and C, D (orange vectors). VPERMI2Q receives two source operands and a mask operand, which is also the destination operand (all are wide registers). Therefore, the mask must be re-set before each VPERMI2Q execution. Finally, we use the VPERMI2Q instruction to calculate the final results (green right bottom vectors). The flow requires 4 loads 8 permutations and 8 mask preparations, with total of 20 instructions per 256-bytes of processed data (e. g., plaintexts). We find this method to be very efficient. Other transposing methods can use the VPGATHERQQ/VPSCATTERQQ or the VPUNPCK instructions, but suffer from high instructions' latency, or need to use more instructions for the same task. Note that the VPUNPCK instruction is recommended for transposing a matrix of size 4×4 with elements of size $4 * 64$-bit, but this is not the case here.

Leveraging the pipeline capabilities efficienly Fast AES computations need to operate on multiple independent blocks in parallel [4], in order to hide the latency of the instrcutions and make the flow depend only on their throughput. The optimal number of blocks is determined by the latency and throughput of the VAES* instructions [17], and the number of available registers. The latency of VAES* is 4 cycles on architecture codename "Skylake" (was 7 cycles on earlier processor generations), and their throughput is 1 cycle. In addition, the AVX512 architecture has 32 zmm registers, which can be used together with the VAES* instructions. For example, in AES-CTR we can allocate 11 registers for the AES round keys, and split the rest among the counters and their the plaintext/ciphertext states ($\sim$ 10 each). Consequently, it is possible to process 10 packets of 4 blocks in parallel, instead of only 8 packets of 1 block, as with the current instructions.

4 AES-GCM and AES-GCM-SIV

AES-GCM [18,19] and AES-GCM-SIV [12,13] are AEAD schemes (AES-GCM-SIV is nonce-misuse resistant). Their encryption flows are outlined in Algorithms 2 and 3. Both modes include AES-CTR encryption (the code is already shown in Fig. 1).

Algorithm 2 AES-GCM encryption [18,19]

```
    Inputs: $K$ (128 bits), $IV$ (96 bits), $A$ (AAD), $M$ (message)
    Outputs: $T$ (tag, 128 bits), $C$ (ciphertext)

procedure AES-GCM($K$, $IV$, $A$, $M$)
    $H$ = AES($K$, $0^{128}$)
    $CTR_0 = IV||0^{31}1$
    for $i = 0, 1, \ldots, v-1$ do
        $CTR_i = CTR[127:32]||((CTR[31:0]+i) \pmod{2^{32}})$
        $C_i$ =AES($K, CTR_i$) $\oplus M_i$
    $T$ = GHASH($H, A, C$) $\oplus$ AES($K, CTR_0$)
    return $C$, $T$
```

Algorithm 3 AES-GCM-SIV encryption [12,13]

```
    Inputs: $K_1$ (128 bits), $K_2$ (128 or 256 bits), $N$ (96 bits), $A$ (AAD), $L_A$ ($A$ length
    in bytes), $M$ (message), $L_M$ ($M$ length in bytes)
    Outputs: $T$ (tag, 128 bits), $C$ (ciphertext)

procedure AES-GCM-SIV($K_1$, $K_2$, $N$, $A$, $L_A$, $M$, $L_M$)
    $Tmp$ = POLYVAL($K_1, A||M||L_A||L_M$)
    $T$ = AES($K_2, 0||(Tmp \oplus N)[126:0]$)
    for $i = 0, 1, \ldots, v-1$ do
        $CTR_i = 1||T[126:32]||((T[31:0]+i) \pmod{2^{32}})$
        $C_i = AES(K_2, CTR_i) \oplus M_i$
    return $C = (C_1, \ldots, C_{v-1})$, $T$
```

We focus on optimizing the universal hashing parts of these algorithms, which are not identical: AES-GCM uses GHASH and AES-GCM-SIV uses POLYVAL.

Both hash functions operate in $\mathbb{F} = \mathbb{F}_{2^{128}}$ (but with different reduction polynomials) and evaluate a polynomial with coefficients $X_1, X_2, \dots, X_s$ (for some s) in $\mathbb{F}$ at some point $H \in \mathbb{F}$ (which is the hash key). As shown in [13]:

$$POLYVAL(H, X_1, X_2, \dots, X_s) = \overline{(GHASH((\overline{H \otimes x}), (\overline{X_1}), (\overline{X_2}), \dots, (\overline{X_s})))}$$

so if suffices to demonstrate the implementation of POLYVAL.

The "Aggregated Reduction" method (see [14]) replaces Hoeren's method with a per-block reduction ($T_i = ((X_i \oplus T_{i-1}) \otimes H) \pmod{Q(x)}$), with a deferred reduction based on pre-computing $t > 0$ powers of H stored in a table (Htbl).

$$\begin{aligned} T_i =& ((X_i \otimes H) \oplus (X_{i-1} \otimes H^2) \oplus \cdots \oplus \\ & (X_{i-(t-1)} + T_{i-t}) \otimes H^t) \pmod{Q(x)} \end{aligned}$$

($\otimes$ is field multiplication; $Q(x)$ is the reduction polynomial).

Fig. 3 compares codes for initializing Htbl. Panel (a) describes the legacy implementation with $t = 8$. Panel (b) describes a vectorized implementation for calculating $t = 4 * 8$ powers of H. Both snippets use the GFMUL function for the field multiplication. Fig. 4 presents GFMUL4 that performs 4 multiplications in parallel. The same code is used for GFMUL and GFMUL2, but over different registers (xmm/ymm). Steps 1-10 perform "Schoolbook" multiplication, and Steps 12-20 perform the reduction (see [14]). An implementation that uses "Aggregated Reduction" should first perform $H \otimes X_t$ as in Fig. 4, Steps 1-5. Then process $H^i \otimes X_{t-i}$, $i = 1, \dots, t-1$ in parallel and accumulate the results. Subsequently, perform the reduction steps 7-20.

(a) Legacy Htbl-init(8)

```
vmovdqu   (H), %xmm0
vmovdqu   %xmm0, %xmm1
.irp i,0,1,2,3,4,5,6
  vmovdqu %xmm0,\i*0x10(Htbl)
  call    GFMUL
.endr
vmovdqu   %xmm0, 7*0x10(Htbl)
ret
```

(b) Vectorized Htbl-init(32)

```
vmovdqu   (H), %xmm0
vmovdqu   %xmm0, %xmm1
vmovdqu   %xmm0, (Htbl)
call      GFMUL
vmovdqu   %xmm0, 0x10(Htbl)
call      GFMUL
vbroadcasti64x2 0x10(Htbl),%ymm1
vmovdqu64 (Htbl), %ymm0
call        GFMUL2
vmovdqu64 %ymm0, 0x20(Htbl)
vbroadcasti64x2 0x30(Htbl),%zmm1
vmovdqu64 (Htbl), %zmm0
.irp i,1,2,3,4,5,6,7
  call        GFMUL4
  vmovdqu64 %zmm0, \i*0x40(Htbl)
.endr
ret
```

Fig. 3. Initializing the HTBL. (a) legacy $t = 8$, (b) vectorized $t = 8 * 4$

```
vpclmulqdq  $0x00 , %zmm1, %zmm0, %zmm2
vpclmulqdq  $0x11 , %zmm1, %zmm0, %zmm5
vpclmulqdq  $0x10 , %zmm1, %zmm0, %zmm3
vpclmulqdq  $0x01 , %zmm1, %zmm0, %zmm4
vpxorq     %zmm4, %zmm3, %zmm3

vpslldq  $8 , %zmm3, %zmm4
vpsrldq  $8 , %zmm3, %zmm3
vpxorq        %zmm4, %zmm2, %zmm2
vpxorq        %zmm3, %zmm5, %zmm5

vpclmulqdq   $0x10 , poly(%rip) , %zmm2, %zmm3
vpshufd      $78 ,     %zmm2, %zmm4
vpxorq        %zmm4, %zmm3, %zmm2

vpclmulqdq   $0x10 , poly(%rip) , %zmm2, %zmm3
vpshufd      $78 ,     %zmm2, %zmm4
vpxorq        %zmm4, %zmm3, %zmm2

vpxorq        %zmm5, %zmm2, %zmm0
ret
```

Fig. 4. The function GFMUL4, performs vectorized $A_1 \otimes A_2 \pmod{Q(x)}$.

In the vectorized implementation we load 4 values from Htbl into each zmm register e. g., zmm(i)= $\{H^{4i}, H^{4i+1}, H^{4i+2}, H^{4i+3}\}$. To multiply the matching values (H^i, X_{t-i}), we first need to reverse their order e. g., $zmm(i) = \{H^{4i+3}, H^{4i+2}, H^{4i+1}, H^{4i}\}$. We do this using the `VSHUFI64X2` instruction: "`vshufi64x2 0x1b, %zmm(i), %zmm(i), %zmm(i)`". Eventually, we end with $T_j = \sum_{\substack{i \equiv j \pmod 4 \\ i=1,\ldots,t}} (H^i \otimes X_{t-i})$, $j = 1, \ldots, 4$. Fig. 5 shows the final aggregation step.

```
vextracti64x4     $1 , %zmm0, %ymm1
vpxor            %ymm1, %ymm0, %ymm0
vextracti128      $1 , %ymm0, %xmm1
vpxor            %ymm1, %xmm0, %xmm0
```

Fig. 5. The vectorized Aggregated Reduction method - Final aggregation.

5 Results

We implemented x86 assembly code for AES-CTR and POLYVAL , using `VAES`* and `VPCLMULQDQ` instructions, pipelining 1 or 8 streams in parallel (the suffix "x8" distinguishes the implementations). To predict the potential improvement on future architectures before real samples are available, we used the Intel Software Developer Emulator (SDE) [20]. This tool allows us to count the number of instructions executed during each of the tested functions. We marked the start/end boundaries of each function with "SSC marks" 1 and 2, respectively.

This is done by executing ”`movl ssc_mark, %ebx; .byte 0x64, 0x67, 0x90`” and invoking the SDE with the flags ”-start_ssc_mark 1 -stop_ssc_mark 2 -mix -icl”. The rationale is that a reduced number of instructions typically indicates improved performance that will be observed on a real processor (although the exact relation between the instructions count and the eventual cycles count is not known in advanced).

Table 1 compares the instructions count in our implementations. The results confirm our prediction that AES algorithms can be sped up by a factor of $3-4$x and that better speedups are expected when operating on larger buffers.

Table 1. Instructions count comparison (lower is better)

Algorithm	PT SIZE (bytes)	Legacy impl.	Vectorized impl.	Ratio
AES-CTR	512	608	178	3.42
AES-CTR	8,192	9,248	2,338	3.96
AES-CTRx8	512	493	150	3.29
AES-CTRx8	8,192	7,453	1,890	3.94
POLYVALx8	4,096	2,816	794	3.55
POLYVALx8	8,192	5,536	1,474	3.76
POLYVALx8	16,384	10,976	2,834	3.87

6 Conclusion

This paper shows how to leverage Intel's new instruction `VPCLMULQDQ` and `VAES*` for accelerating encryption with AES. Our results predict that optimized vectorized AES code can approach the new theoretical bound of 0.16 C/B on forthcoming CPUs, about 4x faster than current implementations. We demonstrated optimized AES-CTR and AES-GCM(-SIV) code snippets that can approach this limit. For serial mode such as AES-CBC, we showed how to optimize code by processing multiple message streams in parallel.

Acknowledgements

This research was supported by: The Israel Science Foundation (grant No. 1018/16); The Ministry of Science and Technology, Israel, and the Department of Science and Technology, Government of India; The BIU Center for Research in Applied Cryptography and Cyber Security, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office; The Center for Cyber Law and Policy at the University of Haifa.

References

1. –: Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf (October 2017)
2. Gueron, S.: Intel® Advanced Encryption Standard (AES) New Instructions Set Rev. 3.01. Intel Software Network (2010)
3. Gueron, S., Kounavis, M.: Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Information Processing Letters **110**(14) (2010) 549 – 553
4. Gueron, S.: Intel's New AES Instructions for Enhanced Performance and Security. In: FSE. Volume 5665., Springer (2009) 51–66
5. Bahajji, Z.A.: Indexing HTTPS pages by default. https://security.googleblog.com/2015/12/indexing-https-pages-by-default.html (Dec 2015)
6. –: Percentage of Web Pages Loaded by Firefox Using HTTPS. https://letsencrypt.org/stats/#percent-pageloads (Jan 2018)
7. Hodjat, A., Verbauwhede, I.: Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. IEEE Transactions on Computers **55**(4) (April 2006) 366–372
8. Mathew, S., Satpathy, S., Suresh, V., Anders, M., Kaul, H., Agarwal, A., Hsu, S., Chen, G., Krishnamurthy, R.: 340 mV #x2013;1.1 V, 289 Gbps/W, 2090-Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt GF(2 4) 2 Polynomials in 22 nm Tri-Gate CMOS. IEEE Journal of Solid-State Circuits **50**(4) (April 2015) 1048–1058
9. Manavski, S.A.: CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In: 2007 IEEE International Conference on Signal Processing and Communications. (Nov 2007) 65–68
10. Patchappen, M., Yassin, Y.M., Karuppiah, E.K.: Batch processing of multi-variant AES cipher with GPU. In: 2015 Second International Conference on Computing Technology and Information Management (ICCTIM). (April 2015) 32–36
11. –: The "Zen" Core Architecture. http://www.amd.com/en/technologies/zen-core (Jan 2018)
12. Gueron, S., Lindell, Y.: GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle Per Byte. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. CCS '15, New York, NY, USA, ACM (2015) 109–119
13. Gueron, S., Langley, A., Lindell, Y.: AES-GCM-SIV: Specification and Analysis. Cryptology ePrint Archive, Report 2017/168 (2017) https://eprint.iacr.org/2017/168.
14. Gueron, S., Kounavis, M.E.: Intel® carry-less multiplication instruction and its usage for computing the GCM mode. White Paper (2010)
15. Drucker, N., Gueron, S.: Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH). (June 2018)
16. –: Intel ®64 and IA-32 architectures software developers manual. Volume 3a: System Programming Guide (September 2015)
17. –: Intel ®64 and IA-32 Architectures Optimization Reference Manual. (June 2016)

18. McGrew, D., Viega, J.: The Galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process **20** (2004)
19. McGrew, D.A., Viega, J.: The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Canteaut, A., Viswanathan, K., eds.: Progress in Cryptology - INDOCRYPT 2004, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 343–355
20. −: Intel®Software Development Emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator