Analysis of SIMD Applicability to SHA Algorithms

O. Aciicmez

***Abstract*— It is possible to increase the speed and throughput of an algorithm using parallelization techniques. Single-Instruction Multiple-Data (SIMD) is a parallel computation model, which has already employed by most of the current processor families. In this paper we will analyze four SHA algorithms and determine possible performance gains that can be achieved using SIMD parallelism. We will point out the appropriate parts of each algorithm, where SIMD instructions can be used.**

I. Introduction

Today the security of a cryptographic mechanism is not the only concern of cryptographers. The heavy communication traffic on contemporary very large network of interconnected devices demands a great bandwidth for security protocols, and hence increasing the importance of speed and throughput of a cryptographic mechanism.

A straightforward approach to improve cryptographic performance is to implement cryptographic algorithms in hardware. Hardware implementation of an algorithm is much faster than the software implementation, however this approach has several drawbacks [1]. Two main disadvantages are variability and cost. Usually a custom hardware is designed for just one algorithm, on the other hand, communication systems need several different algorithms to support all cryptographic mechanisms. Also the cost of a custom hardware including maintenance costs are much higher than those of software.

A far better solution is obtained either by designing a general-purpose cryptographic hardware or by using fast software implementations on general-purpose devices. The former still has some drawbacks such as cost and flexibility. Many approaches are available for designing general-purpose fast cryptographic hardware [2] and fast cryptographic software [3] [4].

In this paper, we focus on how single-instruction multiple-data (SIMD) parallel computation model can improve software cryptographic performance. The SIMD model speeds up the software performance by allowing the same operation to be carried out on multiple data elements in parallel.

Most of the current general-purpose computers employ SIMD architectures. AltiVec extension to PowerPC [5], Intel's MMX technology[6], SSE and SSE2 extensions, Sun's VIS [7] and 3DNow! of AMD [8] are examples of currently used SIMD technologies.

We will use Secure Hash Algorithm (SHA) [9][10] as the cryptographic algorithm in this paper. We analyze the possibility to improve current implementation of SHA algorithm by using SIMD architecture and parallelization techniques. We choose Intel Architecture [11][12][13] as the base SIMD platform since it is the most widely used architecture among the ones cited above.

The remainder of the paper is organized as follows: In section 2 and 3, we introduce SIMD concept and the SIMD architecture of Intel including MMX technology and SSE extensions. Section 4 describes SHA algorithm and Section 5 discusses the possible improvements on SHA performance that can be achieved by using SIMD instructions.

II. SIMD Parallel Processing

Single-instruction multiple-data execution model allows several data elements to be processed at the same time. The conventional scalar execution model, which is called single-instruction single-data (SISD) deals only with one pair of data at a time. The programs using SIMD instructions can run much faster than their scalar counterparts. However SIMD enabled programs are harder to design and implement.

The most common use of SIMD instructions is to perform parallel arithmetic or logical operations on multiple data elements. In order to perform parallel SIMD operations, the program must do:

1 . Load multiple data values into SIMD registers.
2 . Perform the SIMD operation on these registers.
3 . If required, load the results to memory.
4 . If more data has to be processed, repeat the steps.

SIMD instructions have the potential to speed-up the software, however there are mainly 2 problems with SIMD model:

1 . If the data layout does not match the SIMD requirements, SIMD instructions may not be used or data rearrangement code is necessary
2 . In case of unaligned data the performance will suffer dramatically.

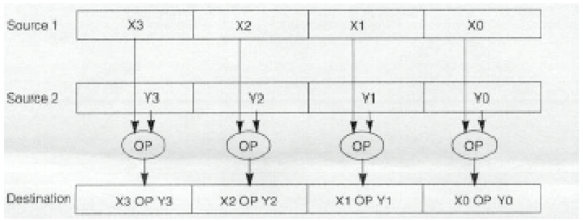

Fig. 1. SIMD Execution Model

III. INTEL'S SIMD ARCHITECTURE

Intel has introduced three extensions into IA-32 architecture to allow IA-32 processors to perform SIMD operations since the production of Pentium II and Pentium with Intel MMX technology processor families. These extensions are MMX technology, SSE extensions, and SSE2 extensions. They provide a group of SIMD instructions that operate on packed integer and/or packed floating point data elements contained in the 64-bit MMX or the 128-bit XMM registers.

Intel introduced MMX Technology in Pentium II and Pentium with MMX Technology processor families. MMX instructions use 64-bit MMX registers and perform SIMD operations on packet byte, word, or doubleword integers located in those registers.

The SSE SIMD integer instructions are the extension of MMX technology. They were introduced in Pentium III processors. These instructions use 128-bit XMM registers in addition to MMX registers and they operate on packed single-precision floating point values contained in the XMM registers and on packed integers contained in the MMX registers.

The latest SIMD extensions of Intel, SSE2, were introduced in the Pentium 4 and Intel Xeon processors. These instructions use both MMX and XMM registers and perform operations on packed double-precision floating-point values and on packed integers. The SSE2 SIMD integer instructions extend IA-32 SIMD operations by adding new 128-bit SIMD integer operations and extending all the 64 bit-SIMD integer operations introduced in the MMX technology and SSE to operate on data contained in the 128-bit XMM registers

The MMX Technology, SSE extensions, and SSE2 extensions provide a rich set of SIMD operations that operates on both integer and floating-point data arrays and on streaming integers and floating point data. These operations can greatly increase the performance of applications running on the IA-32 processors.

In this paper, we are interested in SIMD operations that can be performed on integers. As most of the other cryptographic algorithms, SHA uses integer data and performs operations on integers.

IV. SECURE HASH ALGORITHM (SHA)

SHA is an iterative one-way hash function that can process a message to produce a message digest. There are four different versions of SHA, namely SHA-1, SHA-256, SHA-384, and SHA-512. These four algorithms mainly differ in the number of bits of security that they provide. They further differ in terms of the size of the blocks and words of data that are used during hashing.

Table 1: Secure Hash Algorithm Properties. (values are given in bits)

Algorithm	SHA 1	SHA 256	SHA 384	SHA 512
Message Size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block Size	512	512	1024	1024
Word Size	32	32	64	64
Message Digest Size	160	256	384	512
Security	80	128	192	256

A. Operations

The following operations are applied to w-bit words in all four secure hash algorithms, where w is 32 for SHA-1 and SHA256 and it is 64 for SHA-384 and SHA-512.

1 . Bitwise logical word operations: AND ($\vee$), OR($\wedge$), XOR($\oplus$), and NOT($\neg$)

2 . Addition modulo 2^w.

$$Z = (X + Y) mod 2^w$$

3 . The right shift operation $SHR_n(x)$, where x is a w-bit word and n is an integer with $0 \leq n < w$, is defined by

$$SHR_n(x) = x >> n$$

4 . The rotate right (circular right shift) operation $ROTR_n(x)$ and the rotate left (circular left shift) operation $ROTL_n(x)$, where x is a w-bit word and n is an integer with $0 \leq n < w$, is defined by

$$ROTR_n(x) = (x >> n) \wedge (x << w - n)$$
$$ROTL_n(x) = (x << n) \wedge (x >> w - n)$$

Intel's SIMD architecture provides appropriate instructions for each operation described above. Thus SHA algorithm is fully SIMD-compatible in terms of its operations, and can be implemented in Intel's SIMD architecture.

B. Algorithms

Each algorithm has two stages: preprocessing and hash computation. Preprocessing involves padding a message, setting initialization values to be used in the hash computation, and parsing the padded message into m-bit blocks where m is 512 for SHA-1 and SHA-256 and 1024 for SHA-384 and SHA-512. The hash computation generates a message schedule from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The message digest is the final hash value generated by the hash computation.

For SHA-1 and SHA-256, each message block has 512 bits that are represented as a sequence of sixteen 32-bit words. These two hash computation algorithms perform operations on 32-bit words.

For SHA-384 and SHA-512, each message block has 1024 bits, which are represented as a sequence of sixteen 64-bit words. The operations are performed on 64-bit words in these hash computation algorithms.

Each algorithm employs several rounds with different functions to digest a message block and repeats these computations for each block.

1) SHA-1: At first, the message M is parsed into 16-word blocks M_1, M_2, ... , M_n.

The processing of each M_i involves 80 rounds. Before any of the blocks is processed, the H_j are initialized to some constant values. To process M_i, we proceed as follows:

```
1:  Divide $M_i$ into 16 words $W_0, W_1, ..., W_{15}$
        where $W_0$ is the left-most word
2:  For t = 16 to 79
        $W_t = ROTL_1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$
3:  Let $A = H_0, B = H_1, C = H_2, D = H_3, E = H_4$
4:  For t = 0 to 79 do
        $TEMP = ROTL_5(A) + f_t(B, C, D) + E + W_t + K_t$
        $E = D$
        $D = C$
        $C = ROTL_{30}(B)$
        $B = A$
        $A = TEMP$
5:  $H_0 = H_0 + A$
    $H_1 = H_1 + B$
    $H_2 = H_2 + C$
    $H_3 = H_3 + D$
    $H_4 = H_4 + E$
```

The first two steps are message scheduling and the last two ones are compression function steps. The message digest is the 160-bit string represented by the 5 words $H_0H_1H_2H_3H_4$ calculated after processing the last message block.

The functions used in SHA-1 are shown in Table.

Table 1: Functions used in SHA-1.

Round t	$Function f_t(x, y, z)$
0-19	$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
20-39	$Parity(x, y, z) = x \oplus y \oplus z$
40-59	$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
60-79	$Parity(x, y, z) = x \oplus y \oplus z$

2) SHA-256: At first, the message M is parsed into 16-word blocks $M_1, M_2, ..., M_n$. The processing of each M_i involves 64 rounds. Before processing any blocks, the H_j are initialized to some constant values. To process M_i, we proceed as follows:

```
1:  Divide $M_i$ into 16 words $W_0, W_1, ..., W_{15}$
        where $W_0$ is the left-most word
2:  For t = 16 to 63
        $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
3:  Let $A = H_0, B = H_1, C = H_2, D = H_3$
        $E = H_4, F = H_5, G = H_6, H = H_7.$
4:  For t = 0 to 63 do
        $T_1 = H + \sum_1(E) + Ch(E, F, G) + W_t + K_t$
        $T_2 = \sum_0(A) + Maj(A, B, C)$
        $H = G$
        $G = F$
        $F = E$
        $E = D + T_1$
        $D = C$
        $C = B$
        $B = A$
        $A = T_1 + T_2$
5:  $H_0 = H_0 + A$
    $H_1 = H_1 + B$
    $H_2 = H_2 + C$
    $H_3 = H_3 + D$
    $H_4 = H_4 + E$
    $H_5 = H_5 + F$
    $H_6 = H_6 + G$
    $H_7 = H_7 + H$
```

Similar to SHA-1, the first two steps are again message scheduling and the last two steps are compression function steps. The message digest is the 256-bit string represented by the 8 words $H_0H_1H_2H_3H_4H_5H_6H_7$ calculated after that the last message block is processed.

The functions used in SHA-256 are shown in Table.

Table 2: Functions used in SHA-256.

Function	Definition
$Ch(x, y, z)$	$(x \wedge y) \oplus (\neg x \wedge z)$
$Maj(x, y, z)$	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\sum_0(x)$	$ROTR_2(x) \oplus ROTR_{13}(x) \oplus ROTR_{22}(x)$
$\sum_1(x)$	$ROTR_6(x) \oplus ROTR_{11}(x) \oplus ROTR_{25}(x)$
$\sigma_0(x)$	$ROTR_7(x) \oplus ROTR_{18}(x) \oplus SHR_3(x)$
$\sigma_1(x)$	$ROTR_{17}(x) \oplus ROTR_{19}(x) \oplus SHR_{10}(x)$

3) SHA-512 and SHA-384: These algorithms are very similar to SHA-256. They use 64-bit words and require 80 rounds to process one message block.

At first, the message M is parsed into 16-word blocks $M_1, M_2, ..., M_n$. The processing of each M_i involves 80 rounds. Before processing any blocks, the H_j are initialized to some constant values. To process M_i, we proceed as follows:

```
1:  Divide $M_i$ into 16 words $W_0, W_1, ..., W_{15}$
        where $W_0$ is the left-most word
2:  For t = 16 to 79
        $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
3:  Let $A = H_0, B = H_1, C = H_2, D = H_3$
        $E = H_4, F = H_5, G = H_6, H = H_7.$
4:  For t = 0 to 79 do
```

```
        $T_1 = H + \sum_1(E) + Ch(E,F,G) + W_t + K_t$
        $T_2 = \sum_0(A) + Maj(A,B,C)$
        $H = G$
        $G = F$
        $F = E$
        $E = D + T_1$
        $D = C$
        $C = B$
        $B = A$
        $A = T_1 + T_2$
5:    $H_0 = H_0 + A$
      $H_1 = H_1 + B$
      $H_2 = H_2 + C$
      $H_3 = H_3 + D$
      $H_4 = H_4 + E$
      $H_5 = H_5 + F$
      $H_6 = H_6 + G$
      $H_7 = H_7 + H$
```

The first two steps are the message scheduling steps and the last two are compression function steps. The message digest for SHA-512 is the 512-bit string represented by the 8 words $H_0H_1H_2H_3H_4H_5H_6H_7$ calculated after processing the last message block. The message digest for SHA-384 is the 384-bit string represented by the 6 words $H_0H_1H_2H_3H_4H_5$ calculated after that the last message block is processed.

The functions used in SHA-384 and SHA-512 are shown in Table.

Table 3: Functions used in SHA-384 and SHA-512.

Function	Definition
$Ch(x,y,z)$	$(x \wedge y) \oplus (\neg x \wedge z)$
$Maj(x,y,z)$	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\sum_0(x)$	$ROTR_{28}(x) \oplus ROTR_{34}(x) \oplus ROTR_{39}(x)$
$\sum_1(x)$	$ROTR_{14}(x) \oplus ROTR_{18}(x) \oplus ROTR_{41}(x)$
$\sigma_0(x)$	$ROTR_1(x) \oplus ROTR_8(x) \oplus SHR_7(x)$
$\sigma_1(x)$	$ROTR_{19}(x) \oplus ROTR_{61}(x) \oplus SHR_6(x)$

V. Applicability of SIMD Operations to SHA

In this work, we are interested in two types of parallelism that can be achieved using INTEL SIMD instructions: parallelism within a thread and thread-level parallelism. We use the word "thread" to mean the ensemble of all the operations to hash one file. Hashing more than one file simultaneously is referred as thread-level parallelism. The purpose of parallelism within a thread is to speed up hashing a file by performing SIMD applicable operations on several data elements at the same time.

In order to perform the same operation on different data simultaneously, the values that the operation uses should be known in advance. The level of parallelism depends on how early the values to be used are known. The level of parallelism is the number of operations that can be executed together.

In this section, we will analyze SIMD applicability of each SHA algorithm and achievability of the two types of parallelism.

A. Thread-level parallelism

It requires appropriate SIMD instructions for each operation used in SHA algorithms to implement thread-level parallelism. Fortunately, INTEL's SIMD architecture contains all the required SIMD instructions to perform all operations of SHA algorithms.

SHA-1 and SHA-256 perform operations on 32 bit words. 64 bit MMX registers can store two 32 bit words, so we can hash any two files simultaneously using one of these two algorithms. Moreover, we can also hash four files at the same time if 128 bit XMM registers are used.

SHA-384 and SHA-512 perform operations on 64 bit words. If MMX registers are used, no parallelism can be achieved since an MMX register can only hold one 64 bit word. However we can obtain a high performance gain by using MMX registers instead of 32 bit general purpose registers. We can hash two files in parallel by using XMM registers.

In both cases more files can be hashed using XMM registers, which is expected due to the size of the registers.

B. Parallelism within a thread

To speed up hashing on a file, we have to combine same operations of different rounds and use SIMD instructions to perform these operations at a time. In order to combine the same operation of two consecutive rounds, we must know the values that will be used in the next round while we are processing the round before. If we know these values in advance, we can successfully convert these operations into one SIMD instruction.

We had to analyze each operation of SHA algorithms to determine whether this operation is SIMD applicable or not. The rest of this section introduces the results of our analysis. The following subsections give the SIMD applicable operations of each SHA algorithm.

1) SHA-1: There are five main parts of the algorithm that SIMD instructions can be used. Most important of them is message scheduling.

$$W_t = ROTL_1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$$

To be able to compute W_t, we first have to compute W_{t-3}. That is the reason why W_t's of only three consecutive rounds can be computed simultaneously. However the computation of

$$W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$$

can be executed 8 at a time. On the other hand, the maximum number of parallel operations that can be executed is 4 due to the restrictions of INTEL architecture. If MMX registers or XMM registers are used, we can perform two 32-bit operations or four 32-bit operations, respectively. So the maximum level of parallelism that can be reached is 4.

Another part of the algorithm we can apply SIMD instructions is the summation: $E + W_t + K_t$.

After completing message scheduling, we can perform the addition $W_t + K_t$ of all rounds at the same time. But again we are restricted by 4 operations at a time. The first addition requires the value of E, which is determined by $ROTL_{30}(B)$ operation executed three rounds before. Therefore, three summations can be performed together simultaneously.

Other SIMD applicable operations are the computations of $f_t(B, C, D)$ and $ROTL_{30}(B)$. The values of B and D of one round are the same as the values of A and C of the previous round, respectively. The value of C used to calculate $f_t()$ is the result of $ROTL_{30}(B)$ operation of the previous round. This gives us the opportunity to use SIMD instructions for calculations of $f_t(B, C, D)$ and $ROTL_{30}(B)$. If we want to compute two $f_t(B, C, D)$ operations in parallel, we first need to compute two $ROTL_{30}(B)$ operations in parallel. If SIMD instructions are not used to calculate $ROTL_{30}(B)$, we can still fasten the computation of $f_t(B, C, D)$ by using SIMD for the parts that only B and D are involved. In this case, it is more efficient to use the below Boolean equation as Maj(x,y,z):

$$Maj(X, Y, Z) = [Y \vee (X \oplus Z)] \wedge (X \vee Z)]$$

Table 4: SIMD applicable operations of SHA-1 and levels of parallelism of these operations

Operation	LoP
$W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$	4
$ROTL_1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$	3
$W_t + K_t$	4
$E + W_t + K_t$	3
$f_t(B, C, D)$	2
$ROTL_{30}(B)$	2

2) SHA-256: Because SHA-256 uses 32-bit words, we can perform 2 operations in one SIMD instruction if MMX registers are used, or we can perform 4 operations if we use XMM registers.

The most important part of SHA-256 that SIMD instructions can be succesfully mounted is message scheduling.

$$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$$

To be able to compute W_t, we first have to compute W_{t-2}. Because of it, we can just compute W_t of two consecutive rounds simultaneously. But the computation of

$$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$$

can be fasten 4 times using XMM registers.

SIMD instructions can also be used in the summation $H + W_t + K_t$. After completing message scheduling, we can perform the addition $W_t + K_t$ of all rounds at the same time. But we are restricted by 4 operations at a time. The first addition requires the value of H, which is determined by $D + T_1$ operation executed four rounds before. Therefore, four summations can be performed together simultaneously.

Table 5: SIMD applicable operations of SHA-256 and levels of parallelism of these operations.

Operation	LoP
$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	4
$\sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$W_t + K_t$	4
$H + W_t + K_t$	4

3) SHA-382 and SHA-512: These algorithms use 64-bit words, so maximum level of parallelism that can be achieved is two when XMM registers are used. Because the structures of these algorithms and SHA-256 are same, the SIMD applicable parts of these three algorithms are also same. The only difference is the maximum level of parallelism that can be achieved, which is only 2 for SHA-384 and SHA-512.

Table 6: SIMD applicable operations of SHA-382 and SHA-512 and levels of parallelism of these operations.

Operation	LoP
$W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$\sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$	2
$W_t + K_t$	2
$H + W_t + K_t$	2

VI. Conclusion and Future Work

In this paper, we introduced the SIMD technology of INTEL IA-32 processor family and analyzed the possible application of this technology to four different SHA algorithms. We showed that each SHA algorithm has a great potential to boost both its speed and throughput using SIMD technology. As the future work, the modifications pointed out in this paper can be implemented and the performance gains obtained by each modification can be analyzed.

References

[1] Internet Society Symposium on Network and Distributed System Security, *Parallelized Network Security Protocols*, 1996.

[2] Rainer Buchty, *Cryptonite - A Programmable Crypto Processor Architecture For High-Bandwidth Applications*, Ph.D. thesis, Technische Universitt Mnchen, December 2002.

[3] Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems, *Towards High Performance Cryptographic Software*, 1995.
[4] Antoon Bosselaers, René Govaerts, and Joos Vandewalle, “Fast hashing on the Pentium,” *Lecture Notes in Computer Science*, vol. 1109, pp. 298–??, 1996.
[5] “Altivec,” http://e-www.motorola.com/webapp/sps/site/.
[6] David Bistry, Carole Delong, Dr. Mickey Gutman, et al., *The Complete Guide to MMX Technology*, McGraw - Hill Inc., 1997.
[7] “Vis,” http://www.sun.com/processors/vis/.
[8] “3dnow!,” http://www.amd.com/us-en/Processors/TechnicalResources/.
[9] National Institute of Standards and Technology, *Specifications for the SECURE HASH STANDARD*, April 1995.
[10] National Institute of Standards and Technology, *Specifications for the SECURE HASH STANDARD*, August 2002.
[11] Intel Corporation, *IA-32 Intel Architecture Software Developer’s Manual Volume 1*, 2003.
[12] Intel Corporation, *IA-32 Intel Architecture Software Developer’s Manual Volume 2*, 2003.
[13] Intel Corporation, *IA-32 Intel Architecture Optimization*, 2003.
[14] Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison Wesley Publishing Company, 1995.
[15] Hans-Peter Messmer, *The Indispensable Pentium Book*, Addison Wesley Publishing Company, 1995.
[16] Kevin R. Wadleigh and Isom L. Crawford, *Software Optimization for High Performance Computing*, Prentice Hall PTR, 2000.
[17] Rick Booth, *Inner Loops, A Sourcebook for Fast 32-bit Software Development*, Addison Wesley Publishing Company, 1997.
[18] Intel Corporation, *IA-32 Intel Architecture Software Developer’s Manual Volume 3*, 2003.
[19] Intel Corporation, *Desktop Performance and Optimization for Intel Pentium 4 Processor*, Feb. 2001.
[20] Intel Corporation, *Intel Architecture Optimization*, Feb. 1999.
[21] B. Dixon and A. K. Lenstra, “Factoring integers using simd sieves,” *Lecture Notes in Computer Science*, vol. 765, pp. 28–39, 1994.
[22] Antoon Bosselaers, René Govaerts, and Joos Vandewalle, “SHA: A design for parallel architectures?,” *Lecture Notes in Computer Science*, vol. 1233, pp. 348–??, 1997.
[23] Eric C. Seidel, “Tomorrows cryptography: Parallel computation via multiple processors, vector processing, and multi-cored chips,” December 2002.
[24] Raghav Bhaskar, Pradeep K. Dubey, Vijay Kumar, Atri Rudra, and Animesh Sharma, “Efficient galois field arithmetic on simd architectures,” .
[25] Atri Rudra, Vijay Kumar, Pradeep K. Dubey, Raghav Bhaskar, and Animesh Sharma, “Data-sliced implementation of reed-solomon and rijndael on simd processors,” .
[26] Atri Rudra, Pdadeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi, “Efficient implementation of rijndael encryption with composite field arithmetic,” .