IMPLEMENTATION AND PERFORMANCE ANALYSIS OF SEAL ENCRYPTION ON FPGA, GPU AND MULTI-CORE PROCESSORS

***Abstract*— Accelerators are special purpose processors designed to speed up compute-intensive sections of applications. Field programmable gate arrays (FPGAs) and graphics processing units (GPUs) offer scope for hardware acceleration of applications. FPGAs are highly customizable, while GPUs provide massive parallel execution resources and high memory bandwidth. In general, FPGAs provide the best expectation of performance, flexibility and low overhead, while GPUs tend to be easier to program. In this paper, we compare the performance of these architectures, presenting a performance study of SEAL, a fast, software-oriented encryption algorithm on a Virtex-6 FPGA, a Graphics Processor Unit (GPU), and Intel Core i7, a hyper-threaded, quad core multi-core (CMP). We perform a comparative study of application behavior on these three diverse accelerators considering performance and we show that each platform has relative competitive advantages in encrypting an input plaintext using SEAL.**

Keyword- Cryptography, Encryption, FPGA, Reconfigurable Computing, GPU, CMP

I. INTRODUCTION

The demand for efficient cryptographic solutions has been continuously growing in the last decade as a consequence of using the Internet in critical areas like business, government and healthcare. Encryption must often be performed at high data rates, a requirement sometimes met with the help of supporting cryptographic hardware. The computational cost of software cryptography is a function of both the complexity of the algorithm and the quality of its implementation. However, regardless of implementation, a cryptographic algorithm designed to run well in hardware will typically not perform in software as well as an algorithm optimized for software execution. Often what is needed is a well-designed, software-optimized encryption method for today's general purpose computers. To this end, Phil Rogaway and Don Coppersmith designed SEAL (Software Encryption Algorithm) [1].

SEAL is a stream cipher, namely incoming data are streamed into the algorithm and continuously encrypted. Stream ciphers are much faster than block ciphers (Blowfish, IDEA, DES). SEAL is a length increasing pseudo-random encryption algorithm which maps a 32-bit sequence number *n* to a keystream *L* under the control of a 160-bit secret key. During the initialization phase, SEAL preprocesses the key into a set of larger tables using the Secure Hash Algorithm SHA1. These tables are then used to speed up encryption and decryption. SEAL was designed as an algorithm appropriate for software implementation on 32-bit processors with small register files.

SEAL is an alternative algorithm to software-based Data Encryption Standard (DES), Triple DES (3DES), and Advanced Encryption Standard (AES), with a lower impact to CPU execution time.

The SEAL Encryption feature provides support for the SEAL Algorithm in Cisco Internetwork Operating System (IOS) IP Security (IPSec) implementations. Moreover, since SEAL has the ability to generate portions of the keystream without having to restart from the beginning, it is exceptionally fast for encrypting streaming data at high data rates, in applications such as on-the-fly disk I/O encryption.

On the platform architecture front, there has been a major shift towards systems with multiple cores, driven by the limited instruction level parallelism and the prohibitive power dissipation of high frequency, single-threaded / single-core processors. Moreover, reconfigurable logic such as FPGAs, vector processors such as the Synergistic Processing Elements (SPEs) in Cell processor, and Graphics Processing Units (GPUs) have been shown to speed up applications in multimedia, graphics, data mining, scientific computing, etc. by orders of magnitude, compared with conventional, homogeneous multi-cores.

GPUs are particularly good in exploiting fine-grain SIMT (Single Instruction Multiple Threads) parallelism. The application is partitioned in threads that are executed in parallel on the massively parallel computational substrate.

On the other hand, CMPs can handle fewer independent threads, at a coarser granularity. CPU vendors have also added SIMD operations on their products, such as the SSE multimedia ISA extensions for Intel x86, to exploit the vectorization opportunities offered by many applications. The latest conventional CMPs come with up to 6 cores (12 threads for cores with SMT capabilities).

There is little systematic research on how accelerators based on different computing substrates, such as homogeneous and heterogeneous multi-cores, vector accelerators, and reconfigurable devices compare in terms of performance. Our work compares the performance of these architectures, presenting a performance study of SEAL on a high performance Virtex-6 FPGA, the latest Nvidia GPU GeForce GTX 480 based on the Fermi architecture and the quad-core Intel Core i7.

The rest of the paper is organized as follows. Chapter II provides the details of the SEAL algorithm. Chapter III describes the FPGA architecture, implementation and performance analysis. In Chapter IV we present an implementation of SEAL on the GPU architecture and show its performance improvements over the optimized Core i7 implementation. Chapter V describes related work and Chapter **Error! Reference source not found.** concludes our work.

II. DESCRIPTION OF SEAL ALGORITHM

SEAL is a type of cryptographic object called a *pseudorandom function family* [2]. SEAL is a length increasing pseudorandom function that, under the control of a random 160-bit key *a*, expands a 32-bit position index *n* to an *L* bit keystream (Fig.1). Number *L* can be made arbitrarily large as needed for a target application but output lengths ranging from a few bytes to a few thousand bytes are anticipated. In this paper, we assume that the output length *L* is exactly 4 KB (or 2^{10} 32-bit words). The *L*-bit output keystream *y* is used to encrypt input plaintext *X* by using the XOR operation.

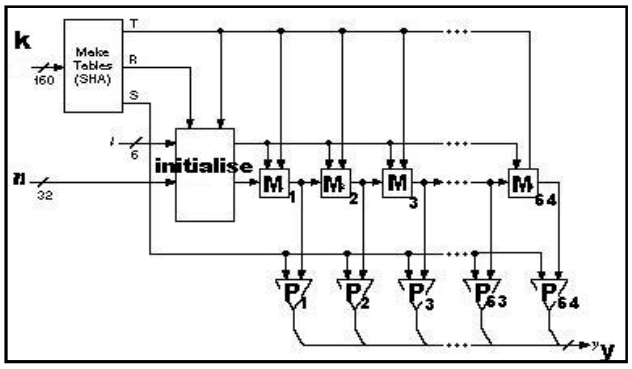

Figure 2. SEAL functional diagram. Output y is the encrypted keystream.

The algorithm is divided into two steps [3]. Step 1 involves *Tables generation.* This step uses the compression function of SHA-1 to expand the secret key a into larger tables *T, S,* and *R*. These tables are fixed and can be precomputed after the key a has been established. Tables *T* and *S* are 2KB bytes and 1KB in size, respectively. The size of table *R* depends on the desired bit length *L* of the keystream – each 1KB of keystream requires 16 bytes of *R*.

Table generation is typically done once over the course of a communication session. Typically, this session takes substantial amount of time and is not in the critical path. It is acceptable, in most applications, to spend this time to map the short key *a* to a longer representation. Therefore, SEAL is not an appropriate choice for applications that require rapid key

function $\mathrm{SEAL}_a(n)$

$y = \lambda$;

for $\ell \leftarrow 0$ **to** ∞ **do**

$\mathrm{Initialize}_a(n, \ell,\ A, B, C, D,\ n_1, n_2, n_3, n_4)$;

for $i \leftarrow 1$ **to** 64 **do**

$P \leftarrow A$ & 0x7fc; $B \leftarrow B + T[P/4]$; $A \leftarrow A) \!) \!) 9$; $B \leftarrow B \oplus A$;
$Q \leftarrow B$ & 0x7fc; $C \leftarrow C \oplus T[Q/4]$; $B \leftarrow B) \!) \!) 9$; $C \leftarrow C + B$;
$P \leftarrow (P + C)$ & 0x7fc; $D \leftarrow D + T[P/4]$; $C \leftarrow C) \!) \!) 9$; $D \leftarrow D \oplus C$;
$Q \leftarrow (Q + D)$ & 0x7fc; $A \leftarrow A \oplus T[Q/4]$; $D \leftarrow D) \!) \!) 9$; $A \leftarrow A + D$;
$P \leftarrow (P + A)$ & 0x7fc; $B \leftarrow B \oplus T[P/4]$; $A \leftarrow A) \!) \!) 9$;
$Q \leftarrow (Q + B)$ & 0x7fc; $C \leftarrow C + T[Q/4]$; $B \leftarrow B) \!) \!) 9$;
$P \leftarrow (P + C)$ & 0x7fc; $D \leftarrow D \oplus T[P/4]$; $C \leftarrow C) \!) \!) 9$;
$Q \leftarrow (Q + D)$ & 0x7fc; $A \leftarrow A + T[Q/4]$; $D \leftarrow D) \!) \!) 9$;
$y \leftarrow y \ \|\ B + S[4i-4] \ \|\ C \oplus S[4i-3] \ \|\ D + S[4i-2] \ \|\ A \oplus S[4i-1]$;
if $|y| \geq L$ **then return** $(y_0 y_1 \ldots y_{L-1})$;
if $odd(i)$ **then** $(A,\ C) \leftarrow (A + n_1,\ C + n_2)$
else $(A,\ C) \leftarrow (A + n_3,\ C + n_4)$;

Figure 3. Cipher mapping 32-bit index *n* to *L*-bit string SEAI-a(n) under the control of tables *T, R,* and *S* [1].

procedure $\mathrm{Initialize}_a(n, \ell,\ A, B, C, D,\ n_1, n_2, n_3, n_4)$

$A \leftarrow n \oplus R[4\ell]$;
$B \leftarrow (n) \!) \!) 8) \oplus R[4\ell + 1]$;
$C \leftarrow (n) \!) \!) 16) \oplus R[4\ell + 2]$;
$D \leftarrow (n) \!) \!) 24) \oplus R[4\ell + 3]$;

for $j \leftarrow 1$ **to** 2 **do**
$P \leftarrow A$ & 0x7fc; $B \leftarrow B + T[P/4]$; $A \leftarrow A) \!) \!) 9$;
$P \leftarrow B$ & 0x7fc; $C \leftarrow C + T[P/4]$; $B \leftarrow B) \!) \!) 9$;
$P \leftarrow C$ & 0x7fc; $D \leftarrow D + T[P/4]$; $C \leftarrow C) \!) \!) 9$;
$P \leftarrow D$ & 0x7fc; $A \leftarrow A + T[P/4]$; $D \leftarrow D) \!) \!) 9$;

$(n_1,\ n_2,\ n_3,\ n_4) \leftarrow (D,\ B,\ A,\ C)$;

$P \leftarrow A$ & 0x7fc; $B \leftarrow B + T[P/4]$; $A \leftarrow A) \!) \!) 9$;
$P \leftarrow B$ & 0x7fc; $C \leftarrow C + T[P/4]$; $B \leftarrow B) \!) \!) 9$;
$P \leftarrow C$ & 0x7fc; $D \leftarrow D + T[P/4]$; $C \leftarrow C) \!) \!) 9$;
$P \leftarrow D$ & 0x7fc; $A \leftarrow A + T[P/4]$; $D \leftarrow D) \!) \!) 9$;

Figure 1. Initialization of *(A, B, C, D , nl , n2, n3, n4)* from *n*. This initialization depends on tables *T* and *R* [1].

set up [1]. In the experimental evaluation section, we will assess performance degradation due to frequent key changes.

The second step is *the pseudorandom function.* Given the number of bits *L*, the tables *T, R,* and *S* (determined by *a*), and a 32-bit position index *n*, the algorithm stretches *n* to an *L*-bit pseudorandom string *y*. The algorithm uses the routine *Initialize* which maps *n* and to the words *A,B,C, D, n1, n2, n3, n4* (Fig.2). These variables are modified over 64 iterations as shown in Fig. 3.

SEAL algorithm uses a few heuristics to improve the capability of the cipher to fend off attacks. For example, it uses a large, secret key-derived S-box (the 2KB table *T*). Moreover, it uses an internal state which does not directly manifest itself in the data stream (the registers n_1, n_2, n_3, n_4 which modify A and C at the end of each iteration).

A. *Parallelism Potential*

The SEAL algorithm can be applied concurrently on successive sections of the input as streaming plaintext *X* becomes available. Thus, thread-level parallelism, i.e. replicating the computation shown in Figure 2, is scalable with the number of cores available and is only limited by the available bandwidth to memory.

On the other hand, there is limited instruction and data-level parallelism at the inner loop of Figure 3, because of the inter-dependences among instructions of the inner loop. However, the outer loop of Figure 3 can be unrolled since all iterations are independent (variables *A, B, C* and *D* are initialized at the beginning of each iteration). We use this observation to exploit SIMD parallelism in platforms with vector processing capabilities.

We expect significant performance improvements not only by exploiting thread- and data-level parallelism, but also by increasing clock frequency and resolving data dependencies quickly. This is manifested in the experimental evaluation section by the competitive performance of Core i7, a high frequency processor with advanced architecture within each 2-way SMT core.

III. FPGA IMPLEMENTATION

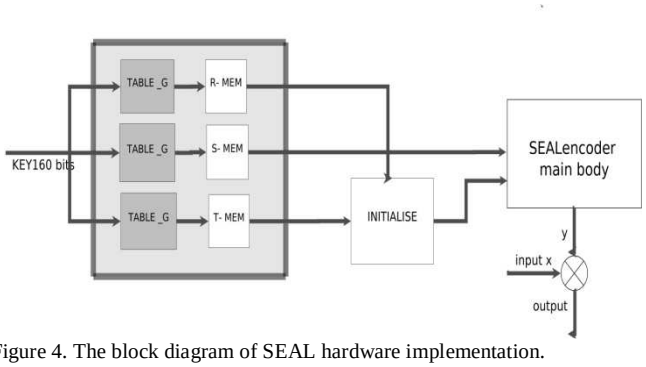

Figure 4. The block diagram of SEAL hardware implementation.

Figure 4 shows the block diagram of the FPGA hardware implementation. We implemented the module *TABLE_G* for the production of the three tables *R, S,* and *T* (Figure 5). This module is just the compression function of the Secure Hash Algorithm SHA-1.

Each time *TABLE_G* runs, it produces a different output of (160 bits), which is written to five positions of memory (5x32 bits). We parallelized the process for the production of the 3 tables - memories as shown in Figure 5. For the implementation of these tables, we used BRAMS 32-bit wide and with sizes of 2KB for *T*, 1KB for *S* and 64 bytes for *R* respectively.

The *Initialize* module uses four 32-bit registers, *A, B, C,* and *D,* whose initial values are determined by *n* and the key-derived tables *R* and *T*. The module maps the 32-bit position index *n* and the iteration counter *l* to eight 32-bit words A_0, B_0, C_0, D_0, n_1, n_2, n_3, n_4.

These registers are modified over several iterations in the *main body* of SEAL encryption to produce A_i, B_i, C_i, D_i on each iteration. In each round nine bits of a register (either A, B, C, or D) are used to index into table *T*. The value retrieved from *T* is then added to or XORed with the contents of a second register: again one of *A, B, C,* or *D* (Figure 3*)*.

The first register is then circularly shifted by nine positions. In some rounds the second register is further modified by adding or XORing it with the (now shifted) first register. When registers A_i, B_i, C_i, and D_i are written, they are added to the keystream, each masked by first adding or XORing it with a certain word from table *S*. The iteration is completed by adding to *A* and *C* additional values dependent on *n, n1, n2, n3, n4*.

Keystream values *y* derived from this procedure are XORed with the plaintext data of memory *X*. We take advantage of the fact that on-chip memories (BRAMs) of high-end FPGAs are dual-ported so that we can overlap reads/writes of the keystream *y*.

An interesting design space exploration exercise is the introduction of pipeline stages in the execution of sequential computations for Table Generation and main encryption (Figure 3). For modules that execute non-critical operations such as Table Generation, we are mostly interested in high clock frequency, since these modules will determine the global clock frequency (we use a single clock in our design). Therefore, these modules are heavily pipelined to increase clock frequency.

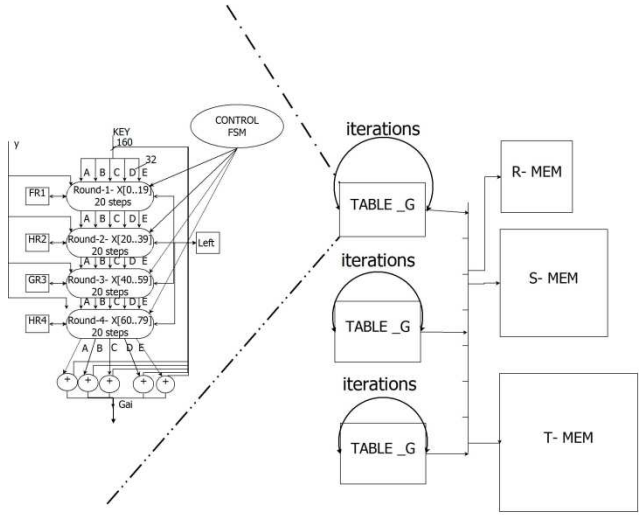

Figure 5 Table Generation Design

A. Experimental Evaluation of FPGA implementation

The FPGA design has been implemented on a Xilinx Virtex-6 HX380T FPGA in Verilog using the Xilinx ISE 12.4 toolset. The hardware complexity of a single accelerator is demonstrated in Table I. A single accelerator processes a 4KB input plaintext message *X* to produce a 4KB encrypted stream. The HX380T FPGA can accommodate up to 32 engines for encrypting 32 4KB plaintext sections in parallel. The 32x accelerator case only replicates the main body of SEAL encryption, not the Table Generation module. Table I also shows that we can achieve 179 MHz clock frequency.

In order to measure the performance of our approach several real-world experiments have been carried out with different configurations of the system and various data-sets (Table II). The first configuration assumes a very frequent key change every 4 KB. This, in effect, modifies the key and re-initializes the tables *T, S,* and *R* at the end of every keystream generation. It is an extreme case used to demonstrate the efficacy of each platform to execute the Table Generation functionality.

The third configuration assumes that the whole input belongs to a single communication session, whereas the second configuration is an intermediate case. Our measurements show that the Table Generation phase is a performance bottleneck if the SEAL user requires frequent secret key modifications.

IV. MULTICORE IMPLEMENTATIONS

One of the objectives of this work is to study the performance of SEAL when fully optimized for both reconfigurable and multi-core platforms. We use the following

TABLE I. RESOURCE UTILIZATION AND MAXIMUM CLOCK FREQUENCY FOR A SINGLE AND 32 SEAL ACCELERATORS, IMPLEMENTED IN A VIRTEX-6 HX380T FPGA.

	Single Accelerator	**32x Accelerators**	**FPGA Total Available**
Logic Slices	1350	46,596	59,760
BRAMs (36Kb)	5	160	768
CLK (MHz)	185	179	

TABLE II FPGA PERFORMANCE MEASUREMENTS

	1 GB encryption with key change every 4KB	**1 GB encryption with key change every 128MB**	**1 GB encryption with key change every 1GB**
Single Accelerator	20.152sec	8.165sec	8.164sec
32x Accelerators	0.629sec	0.255sec	0.255sec

platforms (besides the Virtex-6 FPGA):

- An Intel-based workstation using the Intel Core i7 870 processor (45nm), clocked at 2.93 GHz with 8GB DDR3 memory. This processor integrates four identical cores each with private L1 and L2 caches (32KB and 256 KB, respectively), and a common 8MB L3 Cache.
- An NVIDIA GeForce GTX-480 high-end GPU (40nm), clocked at 1.4 GHz with 1.5 GB of GDDR5 device memory. This GPU is based on the Fermi architecture and includes 480 cores organized in 15 Streaming Multiprocessors (SM) of 32 cores each. Compared with previous GPU generations, it adds an L1/L2 cache hierarchy to the memory architecture to reduce memory access latency and improve programmability. GeForce GTX-480 is connected to Core i7 motherboard via a 16x PCI express bus.

A first, generic optimization with respect to the reference SEAL implementation was to indentify *key* invariant computations in Table Generation and remove them from the critical path of Table Generation, i.e. perform them only once, on program initialization. The benefits are obvious in the common real-world case where a large message is encoded using multiple keys, thus requiring multiple calls to Table Generation.

A. x86 Parallelization

The parallelization on x86 was a two step process: we first created a vectorized (SIMD) version of the algorithm and then exploited multithreading.

We introduced vectorization in both the phases of Table Generation and encryption. Loops without data dependencies (outer loop of Figure 2) are unrolled by a factor of 4, therefore data are processed in group of 128-bit (as vectors of 4 x 32-bit elements), using intrinsics from the SSE2 instruction set.

In order to facilitate the efficient exploitation of the vector capabilities of the processor, data had to be reorganized in memory – either by changing their shape or by padding arrays. Consecutive elements within an input or output vector need to be stored in consecutive, properly aligned addresses in memory.

Multithreading is applied during data encryption, at the granularity of a block (group) of messages. In scenarios where multiple messages need to be encoded, this can be done in parallel, provided that simultaneously encoded messages use the same *T*, *S*, and *R* arrays.

We used a supervisor-worker threads scheme. The supervisor reads a chunk of 4KB plaintext messages from the input and partitions them to message blocks, which are in turn distributed to worker threads. Moreover, if the key needs to change throughout the encoding of the input data, the supervisor executes Table Generation to produce the new same *T*, *S*, and *R* arrays. Finally, the supervisor notifies workers whenever there is no more input data to process, and they – in turn – terminate gracefully.

Unfortunately there are no exploitable opportunities for multithreading in Table Generation, due to the tight data dependencies between successive iterations of the outer loop. However, Table Generation is pipelined with data encryption, whenever multiple keys are used for the encoding of a large data set, resulting to the generation of multiple sets of tables. When workers are encoding using version *i* of the *T*, *S* and *R* tables, the supervisor thread generates their next (*i+1*) version, using the next key. As was discussed in Section II, in realistic situations multiple keys can used successively, however key changes occur at a very low rate that allows Table Generation to fully overlap with data encryption, without becoming a bottleneck. In order to ensure that each message block is encoded with the correct arrays we apply double buffering. There are two copies of *T*, *S* and *R* tables: one written by the supervisor when preparing the tables for the next key, and one read by the workers, generated by the currently effective key. A global barrier synchronizes the supervisor and workers at key change points. At those points, the roles of the two copies of the tables are flipped.

Multithreading has been implemented using the POSIX threads standard. We have experimented with up to 8 worker threads, in order to exploit the 4 cores and the 2-way SMT (Hyperthreading) capabilities of the Core i7 processor.

B. GPU (CUDA) Implementation

A distinguishing characteristic of GPUs is that they are able to manage parallelism at a very fine granularity. Given that they support extremely fast context switching between thread *warps* (i.e. groups of threads) upon stalls of any kind, abundant parallelism must be available in order to effectively hide the latency introduced by stalls and keep GPU utilization high. Another interesting feature of GPUs – especially the latest Nvidia Fermi architecture – is that they allow the configuration of on-chip cache memory as either software- or hardware-controlled. It should also be noted that the GPU is an independent device and does not have direct access to the system's main memory. Instead, input data and results must be programmatically transferred between system's main memory and the GPU device memory (GDDR5 DRAM)[1]. Those transfers suffer severe latency and are limited by the PCIe bus bandwidth.

We implemented SEAL on the GPU using CUDA, a programming model by Nvidia, specifically designed and implemented to support general purpose computations on GPUs.

Given that the parallelism in Table Generation is rather limited, Table Generation proved to perform better on the CPU than on the GPU, even after taking into account the cost

[1] Latest Nvidia Fermi GPUs can make this process somewhat transparent to the programmer, yet with many limitations.

of transferring the generated tables to the GPU. Generating the *T*, *S* and *R* tables in the CPU also provides opportunities of pipelining and overlapping Table Generation with data encryption, as described in the previous section. Moreover, since the tables are read-only by the GPU they can be stored to constant memory – a software controlled, low-latency, high bandwidth, read-only cache in the GPU.

In order to overlap memory transfers between device memory and main memory we used the mechanism of streaming offered by CUDA. Streams are named sequences of data transfers and/or computations. Operations within a stream must be performed synchronously and in order. However, operations across different streams are totally independent and asynchronous. Each stream transfers a 64MB block of messages to the GPU, encodes the messages and transfers the results back to the main memory. The block of messages is processed in parallel by 16,384 threads. The block size is an educated choice that satisfies the tradeoff of low memory requirements – so that concurrently active streams do not overflow any level of the GPU memory hierarchy – offering at the same time high parallelism potential – so that GPU computational resources are efficiently utilized, hiding stalls latency.

The GPU memory management unit performs significantly better if accesses to device memory are coalesced, i.e. if they follow specific patterns. In the SEAL encryption implementation coalesced memory accesses could be achieved by transposing data from the input streams and also results before sending them back to the main memory. The performance benefits of memory coalescing proved enough to justify the extra cost of the two transpose operations, which were implemented using the optimized algorithm available in the CUDA SDK [13]. Some extra reorganization of the algorithm allowed the minimization of the number of high-latency memory transfers, favoring fewer, large transfers instead of more, smaller ones.

IV. EXPERIMENTAL EVALUATION AND COMPARISON

Similar to the FPGA implementation, we decided that it could be useful to test our implementations changing key after 1 message or 32Kbit or 4KB, 32768 messages or 1Gbit or 128MB, 262144 messages or 8Gbit or 1GB. We created a random 1GB input file and ran each of our 4 implementations (Initial version (Single Threaded), SIMD version (Single Threaded), Multithreaded + SIMD version for 2, 4, 8 threads, CUDA) 5 times.

Figure 6 compares the speed-up of SEAL implementations for all three scenaria described in Section III.A compared to the execution time of SEAL code running as a single thread on Core i7. The single-threaded code (corresponding to speed up of 1) requires 1.5 secs for 8 Gbit and 1 Gbit sessions, and 11.28 secs when a session is 4 KB in order to encrypt an input plaintext of 128 MB.

Code optimizations for Core i7 are very successful in improving speed-up as shown in Figure 6. Data level parallelism (SIMDization) is more successful when used in Table Generation and makes a pronounced contribution to speed up when session size is 32 Kbits.

Running threads in multiple cores provides linear speed up which is to be expected since threads are independent. Somewhat surprisingly, hyperthreading (when moving from 4 to 8 cores) provides a remarkable speed up of approximately 42% in the first two scenaria. This is an interesting observation since hyperthreading has been shown to provide much lower performance improvement for various workloads and, sometimes, has an adverse impact on execution time.

The FPGA implementation is easily scalable and its performance depends on the number of available accelerators that can fit in the device. The low performance of a single accelerator is mainly due to the low clock frequency of FPGAs compared to high performance processors, and the limited parallelism within an accelerator. FPGAs perform relatively better when session size is 32 Kbits, because they can offer an efficient implementation of the Table Generation module, which becomes the bottleneck in this usage scenario.

The GPU proved to be an appropriate platform for implementing the SEAL algorithm. When we include the copies of plaintext data and tables from CPU main memory to GPU device memory on execution time, speed up drops considerably, below Core i7 performance. This shows that performance is limited by the peak bandwidth of 16x PCIe. We should note, however, that GeForce GTX-480 is one of the fastest commercially available GPUs. Figure 6 shows that is is difficult to keep all 480 streaming processor from computational starvation when limited by PCIe bandwidth.

V. RELATED WORK

This section provides some references to previous work related to developing cryptography solutions both in dedicated hardware on GPUs and FPGAs.

Several applications and especially in cryptography have been proposed and implemented in hardware. Examples include image processing [5], data mining [6] for FPGAs, and AES encryption [7] on GPUs.

Comparison between FPGAs and GPUs has been proposed for video processing applications [8], and data encryption standards (DES [9], SHA-1[14])

Lin Zhou , and Wenbao Han investigated the implementation and performance of SHA-1 using FPGA and GPU, with the view of comparison their salient features [14] .

The DES encryption results [9] shows to process a single, 64-bit block on FPGA and requires only 83 cycles, while the same operation executed on the GPU requires 5.80 X 10^5 cycles. While the GPU does not support some important operations for this application, the main reason for this disparity is that the GPU requires full utilization to take advantage of the hardware's latency hiding design, and this example far underutilized the processor.

An implementation of SEAL in C encrypted the input plaintext at 6.9 MB/sec on an antiquated PC (50MHz). The same code ran at 15.5 MB/sec on a low end RISC workstation (an SGI Indy, which has a 100 MHz MIPS 4600 Processor) [10].

Bosselaers provided experimental results on the performance of various cryptographic algorithms in [11]. A comparison

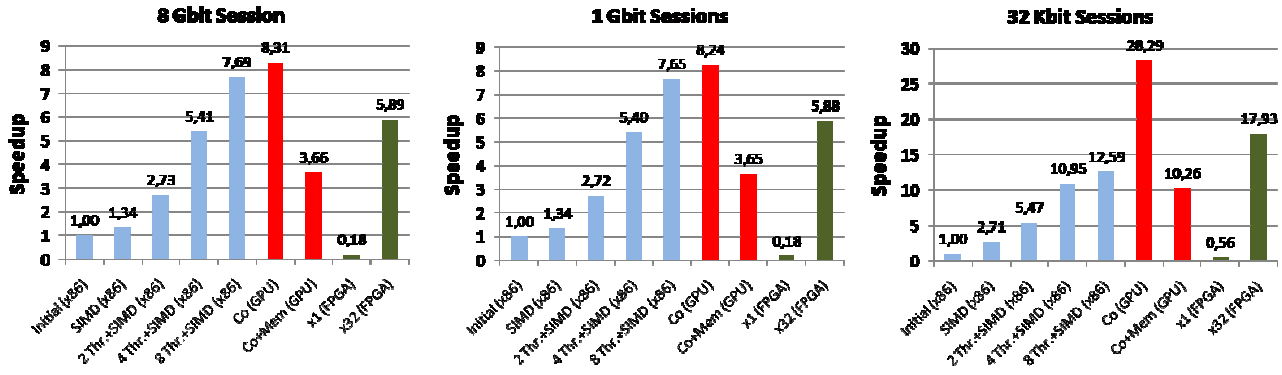

Figure 6. Performance comparison across the three platforms for various configurations

SEAL implementation on the two machines (SPARC and Alpha) is shown in [12].

VI. CONCLUSION

In this paper, we have presented the mapping and optimization of the SEAL Encryption algorithm on an FPGA, an Intel Core i7, and the NVidia GeForce GTX480 GPU. All three platforms were able to exploit the available thread-level parallelism and achieve the high performance. We have found that the modern CMP platforms make better use of the sophisticated hardware-based cache hierarchy as well as high clock frequencies to sustain high utilization of the data path. GPUs have the potential to speed up SEAL algorithm even more provided that they are not limited by the bandwidth of PCIe. Finally, FPGAs can better exploit parallelism of Table Generation module. Thread level parallelism is only limited by the device size and is the main way to alleviate the adverse effects of low clock frequency.

VII. REFERENCES

[1] P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm", Proceedings of the 1993 Cambridge Security Workshop, Springer-Verlag, 1994.

[2] O. Goldreich, S. Goldwasser, S. Micali, "How to construct random functions", Journal of the ACM, Vol 33, No. 4, 1986, pp. 210-217

[3] H. Handschuh, H. Gilbert, "χ^2 cryptanalysis of the SEAL encryption algorithm", Fast Software Encryption, Lecture Notes in Computer Science, Vol. 1267, Springer- Verlag, 1997, pp. 1-12

[4] B. Schneier, Applied Cryptography, Second Edition, John Wiley & Sons, 1996.

[5] B. de Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. In Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pages 93–97, 2006

[6] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines, pages 3–12, 2005.

[7] T. Yamanouchi. AES encryption and decryption on the GPU. GPU Gems 3, July 2007.

[8] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing? In Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, pages 111–118, 2005.

[9] Shuai Che; Jie Li; Sheaffer, J.W.; Skadron, K.; Lach, J. "Accelerating Compute-Intensive Applications with GPUs and FPGAs" In Application Specific Processors, 2008. SASP 2008. Symposium on , pages: 101 - 107 , 8-9 June 2008

[10] P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm", FAST SOFTWARE ENCRYPTION,Lecture Notes in Computer Science, 1994, Volume 809/1994, 56-63.

[11] B. Preneel, V. Rijmen, and A. Bosselaers, Recent developments in the design of conventional cryptographic algorithms, Computer Security and Industrial Cryptography—State of the Art and Evolution,Lecture Notes in Computer Science, Springer-Verlag, Berlin, to appear.

[12] M. Roe, Performance of block ciphers and hash functions—one year later, Fast Software Encryption, Lecture Notes in Computer Science, Vol. 809, Springer-Verlag, Berlin, 1994, pp. 359–362.

[13] NVIDIA CUDA C SDK - Linear Algebra http://developer.download.nvidia.com/compute/cuda/sdk/website/Linear_Algebra.html

[14] Lin Zhou , Wenbao Han, A Brief Implementation Analysis of SHA-1 on FPGAs, GPUs and Cell Processors,. International Conference on Engineering Computation, 2009. ICEC '09 , Hong Kong