Implementation and Performance Analysis of SEAL Encryption on FPGA, GPU and Multi-Core Processors

Kostas Theoharoulis, Charalambos Antoniadis, Nikolaos Bellas and Christos D. Antonopoulos
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
Email: ktheoxaroulis@gmail.com, {haadonia, nbellas, cda}@inf.uth.gr

***Abstract*—Accelerators, such as field programmable gate arrays (FPGAs) and graphics processing units (GPUs), are special purpose processors designed to speed up compute-intensive sections of applications. FPGAs are highly customizable, while GPUs provide massive parallel execution resources and high memory bandwidth. 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 2-way hyper-threaded, 4-core processor. We show that each platform has relative competitive advantages in encrypting an input plaintext using SEAL.**

***Keywords*-Cryptography; Encryption; FPGA; Reconfigurable Computing; GPU; CMP**

I. Introduction

The demand for efficient cryptographic solutions has been continuously growing in the last decade. Encryption must usually be performed at high data rates, a requirement often met with the help of cryptographic hardware. The performance of software cryptography is a function of both the complexity of the algorithm and the quality of its implementation. Typically, algorithms designed and optimized for software implementations, such as SEAL (Software Encryption Algorithm) [1], will outperform software ports of algorithms originally designed for hardware.

SEAL is a stream cipher, i.e. incoming data are streamed into the algorithm and continuously encrypted. The algorithm is supported in Cisco Internetwork Operating System (IOS) IP Security (IPSec) implementations. Moreover, 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 – and Graphics Processing Units (GPUs) have been shown to speed up applications often by orders of magnitude, compared with conventional, homogeneous multi-cores.

There is little systematic research on how accelerators based on different computing substrates 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, an Nvidia GeForce GTX 480 GPU based on the Fermi architecture and the quad-core Intel Core i7.

The rest of the paper is organized as follows. Section II provides the details of the SEAL algorithm. Section III describes the FPGA architecture, implementation and performance analysis. Sections IV and V discuss the parallelization on the GPU and show its performance improvements over the optimized Intel Core i7 implementation. Section VI concludes our work.

II. Description of SEAL Algorithm

SEAL is a pseudo-random function family [2] crypto object. A length increasing pseudo-random function expands a 32-bit position index n to an L bit keystream (Figure 1), under the control of a random 160-bit key . L can be made arbitrarily large ranging, in realistic applications, from a few bytes to a few thousand bytes. In this paper, we assume that the output length L is 4 KB. The L-bit output keystream y is then used to encrypt input plaintext X using the XOR operation.

The algorithm is divided in two steps [3]. Step 1 involves Table generation. This step uses the compression function

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

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

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

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

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

Figure 1. Initialization of A, B, C, D , nl , $n2$, $n3$, $n4$ from n. Initialization is dependent on T and R tables [1].

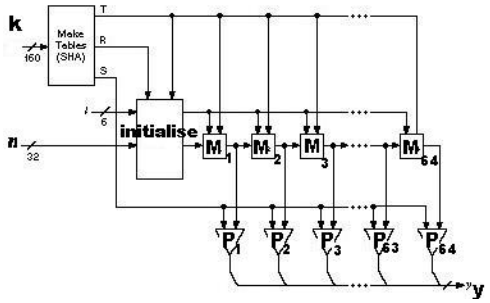

Figure 2. SEAL functional diagram. Output y is the encrypted keystream.

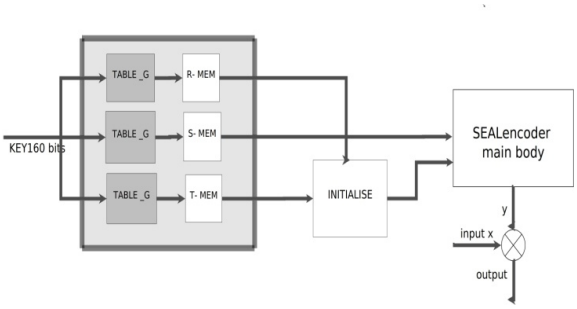

Figure 4. The block diagram of SEAL hardware implementation.

of SHA-1 to expand the secret key into larger tables T, S, and R. These tables are fixed and can be precomputed after the key has been established. Tables T and S are 2KB 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.

Tables generation is typically performed once over the course of a communication session. Although it is computationally expensive, it is not in the critical path for applications that do not require rapid key set up [1]. In the experimental evaluation section, we will assess performance degradation due to frequent key changes.

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

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.

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; $B \leftarrow B + T[P/4]$; $A \leftarrow A) \rangle\rangle 9$; $B \leftarrow B \oplus A$;
    $Q \leftarrow B$ & 0x7fc; $C \leftarrow C \oplus T[Q/4]$; $B \leftarrow B) \rangle\rangle 9$; $C \leftarrow C + B$;
    $P \leftarrow (P + C)$ & 0x7fc; $D \leftarrow D + T[P/4]$; $C \leftarrow C) \rangle\rangle 9$; $D \leftarrow D \oplus C$;
    $Q \leftarrow (Q + D)$ & 0x7fc; $A \leftarrow A \oplus T[Q/4]$; $D \leftarrow D) \rangle\rangle 9$; $A \leftarrow A + D$;
    $P \leftarrow (P + A)$ & 0x7fc; $B \leftarrow B \oplus T[P/4]$; $A \leftarrow A) \rangle\rangle 9$;
    $Q \leftarrow (Q + B)$ & 0x7fc; $C \leftarrow C + T[Q/4]$; $B \leftarrow B) \rangle\rangle 9$;
    $P \leftarrow (P + C)$ & 0x7fc; $D \leftarrow D \oplus T[P/4]$; $C \leftarrow C) \rangle\rangle 9$;
    $Q \leftarrow (Q + D)$ & 0x7fc; $A \leftarrow A + T[Q/4]$; $D \leftarrow D) \rangle\rangle 9$;
    $y \leftarrow y \ \|\ B + S[4i-4] \ \|\ C \oplus S[4i-3] \ \|\ D + S[4i-2] \ \|\ A \oplus S[4i-1]$;
    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 an L-bit string under the control of tables T, R, and S [1].

On the other hand, there is limited instruction and data-level parallelism at the inner loop of Figure 3, because of the inter-dependencies among instructions of the inner loop. However, the outer loop of Figure 3 can be unrolled since all iterations are independent. This allows the exploitation of SIMD parallelism in platforms with such capabilities.

We also expect significant performance improvements 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 depicts 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. 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. For the implementation of these tables, we used 32-bit wide BRAMs, with sizes of 2KB for T, 1KB for S and 64 bytes for R.

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 $A0$, $B0$, $C0$, $D0$, $n1$, $n2$, $n3$, $n4$, which are are modified over several iterations in the main body of SEAL encryption to produce Ai, Bi, Ci, Di on each iteration. 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, thus 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.

A. Experimental Evaluation of FPGA Implementation

The FPGA design has been implemented on a Xilinx Virtex-6 XC6VLX760 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 XC6VLX760 FPGA can accommodate up to 64 engines for encrypting 64 4KB plaintext sections in parallel. The 64x accelerator case only replicates the main body of SEAL encryption, not the Table Generation module. Table I also shows that we can achieve 149 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 (Section V).

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 platforms (besides the Virtex-6 FPGA):

a) 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.

b) 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. Fermi adds an L1/L2 cache hierarchy to the memory architecture to reduce memory access latency and improve programmability. The GTX-480 is connected to the motherboard via a 16x PCI express bus.

A first, generic optimization was to identify key invariant computations in Table Generation and remove them from the critical path, 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.

	1 Accel.	64x Accel.	FPGA Total
Logic Slices	1,450	93,323	118,560
BRAMs (36KB)	5	320	720
Clock (MHz)	158	149	

Table I
Resource utilization and maximum clock frequency for 1 and 64 accelerators, implemented on a Virtex-6 XC6VLX760 FPGA.

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. Multithreading is applied during data encryption, at the granularity of a block (group) of messages. 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. As was discussed in Section II, in realistic situations key changes occur at a very low rate. This allows Table Generation to fully overlap with data encryption, without becoming a bottleneck.

We have experimented with up to 8 worker threads, in order to exploit the 4 cores and the 2-way SMT (Hyper-threading) capabilities of the Core i7 processor.

B. GPU (CUDA) Implementation

GPUs are able to manage parallelism at a very fine granularity. Abundant parallelism must be available in order to effectively hide the latency introduced by stalls and to 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. Moreover, main memory-to-GPU 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. Table Generation proved to perform better on the CPU than on the GPU, given its limited parallelism. Generating the T, S and R tables in the CPU also provides opportunities of pipelining and overlapping Table Generation with data encryption.

Each block of messages is processed in parallel by 16,384 threads. The block size is an educated choice that satisfies the trade-off 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.

GPU upper level caches perform significantly better if concurrent requests do not result to cache bank conflicts. This was achieved by transposing data from the input streams and also results before sending them back to the main memory. 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. In order to overlap memory transfers

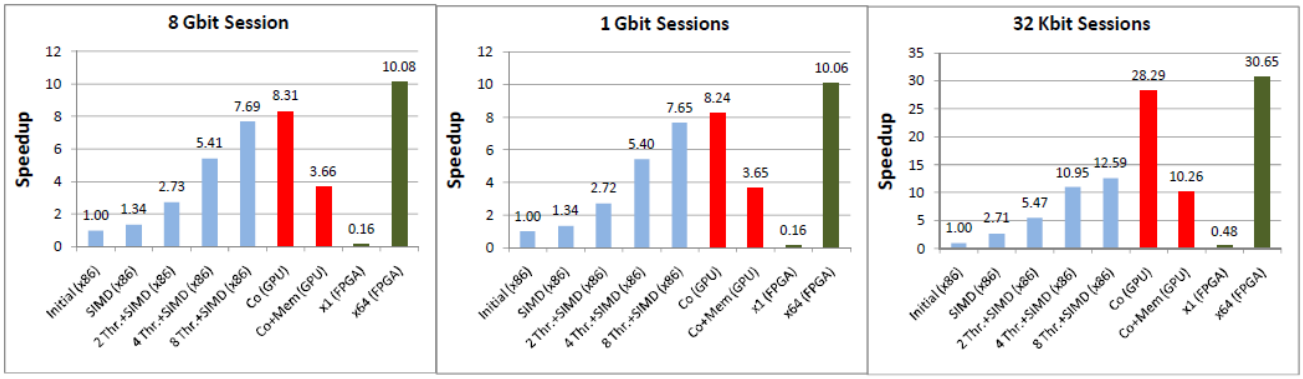

Figure 5. Comparative Performance Evaluation Results.

between device memory and main memory we used the mechanism of streaming offered by CUDA.

V. Experimental Evaluation and Comparison

We tested our implementations changing the key after 1 message (32Kbit), 32768 messages (1Gbit) and 262144 messages (8Gbit). We ran each of our 5 implementations – Initial version (Single Threaded), SIMD version (Single Threaded), Multithreaded + SIMD version (for 2, 4, 8 threads), CUDA and FPGA – on a random 1GBit input file.

Figure 5 compares the speed-up of SEAL implementations for all three scenaria over the execution time of SEAL 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 for 32 KBit sessions.

Code optimizations for Core i7 proved very successful in improving speed-up. 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, as expected, linear speed up. Somewhat surprisingly, hyper-threading (moving from 4 to 8 cores) provides a remarkable speed up of approximately 42% in the first two scenaria. Hyperthreading typically provides much lower performance improvements, however in this case (a streaming application) it proved effective in hiding memory access latency.

The FPGA implementation is easily scalable and manages to outperform all other substrates. Its performance depends on the number of 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. 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 computational part of SEAL (Figure 5, Co GPU). However, when we take memory transfers into account (Co+Mem), performance drops considerably below that of Core i7. Performance is thus 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 5 shows that is is difficult to keep all 480 streaming processors busy given the limited PCIe bandwidth.

VI. Conclusions

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 the 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.

Acknowledgment

This work is partially supported by the EC Marie Curie International Reintegration Grant (IRG) 223819.

References

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

[2] O. Goldreich, S. Goldwasser, and S. Micali, "How to Construct Random Functions," *Journal of the ACM*, vol. 33, no. 4, pp. 210–217, 1986.

[3] H. Handschuh and H. Gilbert, "x^2 cryptanalysis of the seal encryption algorithm," in *"Fast Software Encryption"*, ser. Lecture Notes in Computer Science, Springer-Verlag, Ed., vol. 1267, 1997, pp. 1–17.