Design of a Parallel AES for Graphics Hardware using the CUDA framework

Andrea Di Biagio, Alessandro Barenghi, and Giovanni Agosta
Politecnico di Milano
{dibiagio,barenghi,agosta}@elet.polimi.it

Gerardo Pelosi
Università degli Studi di Bergamo
gerardo.pelosi@unibg.it

Abstract

Web servers often need to manage encrypted transfers of data. The encryption activity is computationally intensive, and exposes a significant degree of parallelism. At the same time, cheap multicore processors are readily available on graphics hardware, and toolchains for development of general purpose programs are being released by the vendors. In this paper, we propose an effective implementation of the AES-CTR symmetric cryptographic primitive using the CUDA framework. We provide quantitative data for different implementation choices and compare them with the common CPU-based OpenSSL implementation on a performance–cost basis. With respect to previous works, we focus on optimizing the implementation for practical application scenarios, and we provide a throughput improvement of over 14 times. We also provide insights on the programming knowledge required to efficiently exploit the hardware resources by exposing the different kinds of parallelism built in the AES-CTR cryptographic primitive.

1. Introduction

Many modern e-banking and e-commerce scenarios need to handle encrypted data to provide secure communications with the users. The encryption and decryption of data is a computationally intensive task, and takes an heavy toll on the computational resources offered by a server. Since the throughput obtainable from a serving endpoint is drastically cut down when using strong cryptographical algorithms to secure the sessions, it is sensible to build dedicated hardware-software accelerators with both an affordable cost and a great ease of integration.

Meanwhile, the aggressive competition for the Graphics Processing Unit (GPU) market is driving these architectures towards increasing levels of hardware parallelism, while containing the costs. Since these platforms are now supported by development toolchains which allow the implementation of general purpose software, they become a prime choice for the implementation of computationally demanding algorithms.

AES is the most popular symmetric key cryptographic algorithms, offering a high degree of parallelism, which can be effectively exploited through natively parallel architectures. Nonetheless, current cryptographic solutions need to be carefully reimplemented in a fashion fit for parallel execution, in order to reach a better cost/performance trade-off with respect to the traditional CPU-based implementations.

In a typical two or three tier enterprise oriented architecture, the most common choice for implementing a cryptographically-enabled first tier is Apache [1] and OpenSSL [2]. Apache has been the most widespread webserver for the last decade [3]. It supports almost every web application environment either directly or through reverse proxy configuration. OpenSSL is the most common implementation of the SSLv2, SSLv3 and TLSv1 transport layer security standards [4] and is recognized as the de-facto standard library to be used in conjunction with Apache.

The binding between the two is realized through a module, aptly named ModSSL [5], which directly maps Apache's cryptographical needs to OpenSSL functions thus allowing the prompt exploit of any advance in the implementation of the underlying toolkit without changing any of the webserver configurations. The typical size of an object transferred during an SSL session is known to range from 35 KB (for HTML-only content) to 150 KB (for text and image content, sent in a single HTTP 1.1 session) [6].

In the remainder of this paper, we address the issue of deploying encryption primitives on the GPU hardware by providing a parallel AES design targeting the NVIDIA GeForce 8 GPU family [7], implementing them within the popular OpenSSL framework.

We explore several parameter choices realizing different implementations and offering insights on the programming perspective that needs to be taken into account when implementing cryptographic algorithms on graphics-oriented hardware.

An extensive experimental campaign supports our implementation choices and points out the trade-offs between the different solutions.

The rest of this paper is organized as follows. Section 2 provides an overview of the AES block cipher and its modes of operation. Section 3 describes the GeForce GPU family, its programming model, and the design of our parallel AES. Section 4 provides an experimental evaluation of the parallel AES implementations, while Section 5 surveys the most closely related works in the field. Finally, Section 6 provide some conclusions and points out some directions for future work.

2. Overview of the AES Block Cipher

The Advanced Encryption Standard (AES) [8] is a symmetric cryptographic algorithm originally requested and adopted by the National Institute of Standards and Technology (NIST) for replacing the Data Encryption Standard (DES) [9]. AES corresponds to a block size restricted version of the Rijndael [10], an iterated block cipher, which can encrypt and decrypt plaintext blocks of size 128 bits using a key size of 128-bit, 192-bit or 256-bit length. The Rijndael cipher was adopted thanks to its ease of implementation on a wide range of 8-bit to 32-bit processing platforms as well as being amenable to high performance ad hoc hardware implementations. Moreover, the clarity and compactness of its design allowed a wide cryptanalytic scrutiny that helped to strengthen the confidence in its security level. In software, AES can be implemented with a fully symmetric structure using only bitwise XOR operations, table-lookups and 8-bit shifts.

AES is well suited to be implemented on processors with a parallel architecture. The cipher is designed for executing a number of round transformations on plaintext where the output of each round is the input of the next one. The number of rounds is determined by the key length: 128-bit uses 10 rounds, 192-bit 12 and 256-bit 14. Each round is composed by the same steps, except for the first round where an extra addition of a round key is added and for the last round where the last step (MixColumns) is skipped. Each step operates on 16 bytes of data (referred as the internal *state* of the cipher) generally viewed as a 4×4 table of bytes or an array of four 32-bit words, where each word corresponds to a column of the state table. The four round stages are *AddRoundKey* (XOR addition of a scheduled round key for blending together the key and the state), *SubBytes* (byte substitution by an S-box, i.e. a lookup table for non-linearity design reasons), *ShiftRows* (cyclical shifting of bytes in each row to realize a inter-word byte diffusion), *MixColumns* (linear transformation which mixes column state data for intra-word inter-byte diffusion).

The different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors having word length of 32 bits or greater [10]. Let us denote with $a_{i,j}$ the generic element of the state table, with $S[256]$ the S-box table and with $\bullet$ a $GF(2^8)$ finite field multiplication [10]. Let T_0, T_1, T_2 and T_3 be four lookup tables containing results from the combination of the aforementioned operations as follows:

$$T_0[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \end{bmatrix} \quad T_1[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \end{bmatrix}$$

$$T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \end{bmatrix} \quad T_3[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \end{bmatrix}$$

These tables are used to compute the round stages operations as a whole as described by the following equation, where k_j is the j-th word of the expanded key and e_j is the j-th column of the state table (seen as a single 32-bit word):

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus k_j$$

The four tables T_0, T_1, T_2 and T_3 (called *T-boxes* from now on) have 256 32-bit word entries each and make up for 4 KB of storage space.

A *KeySchedule* procedure associated to the AES algorithm is responsible for the computation of each round key k_j given the global input key k. In contrast with the round computation, the key expansion operated by the KeySchedule procedure does not expose significant parallelism. However, its result is computed once and used for all the blocks of a given plaintext.

The AES, as any other block cipher, operates on blocks of fixed 128-bit length. Several modes of operation have been standardized to manage the encryption of any plaintext, with arbitrary length [11]. When the length of the plaintext is not a multiple of the block size, it is necessary to add padding to the original message up to a multiple of the block size.

Of the block cipher modes employed for guaranteeing confidentiality, Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter Mode (CTR) are the most popular. The ECB mode is easily parallelizable, since the original plaintext is split into blocks that are independently enciphered with the same key.

However, the ECB mode is not adopted in cryptographic protocols, since identical plaintexts blocks, encrypted with the same key (as would happen when enciphering a file with repeated 16 bytes blocks), lead to the same ciphertext, which is a major leak of secret information that can be exploited by cryptanalytic attacks.

CBC mode is the default choice in current distributions of OpenSSL. In this mode, the sequence of plaintext blocks is enciphered using as input of each block the bitwise XOR between a block of plaintext and the ciphertext obtained from the previous block (or a known initialization vector for the first block).

CTR mode produces the ciphertext as the bitwise XOR between each plaintext block and one of a series of cryptographic pads. The cryptographic pads are obtained through the application of the block cipher to counter initialized with a strong pseudo-randomly generated value and sequentially incremented for each subsequent block.

From a security point of view, CTR mode is considered even safer than CBC [12], [13], thus it has been added in the 1.1 version of the Transport Layer Security protocol standard [14].

3. Parallel Algorithm Design

In recent times, Graphics Processing Units (GPUs) have been considered a potential source of computational power for non-graphical applications, due to the ongoing evolution of their programming interfaces and their appealing cost-performance figures of merit. Recent works had first attempted to adapt "general purpose" applications to the graphic rendering APIs (OpenGL and DirectX), which up to two years ago represented the only interface to tap into the GPU computational resources [15].

The use of GPUs to speed up the computation of AES has been pioneered by D. Cook *et al.* in [16], and further developed by Harrison and Waldron [17]. Both works faced major limitations imposed by GPU hardware and software: on one hand, the GPU instruction set architectures were mostly geared towards floating point computation – thus lacking support for integer and logical operations; on the other hand, GPUs exposed to the programmer a set of operations only mapping typical rendering API, thus resulting unwieldy to program in a general purpose context. However, these limitations are quickly disappearing as GPU designers have been dramatically increasing the level of support for general purpose computing in their platforms [18], [19].

3.1. The NVIDIA G80 Architectures

Modern GPUs include hundreds of processing elements. The NVIDIA G80 GPU series provide a set of independent multi-threaded streaming multiprocessors. Figure 1 shows an overview of the NVIDIA G80 streaming processors array which is the part of the GPU architecture responsible for the general purpose computation. Each streaming multiprocessor is composed by a set of eight streaming processors, two special functional units and a multithreaded instruction issue unit (respectively indicated as SP, SFU and MT-Issue in Figure 1). A SP is a fully pipelined single-issue processing core with two ALUs and a single floating point unit (FPU). SFUs are dedicated to the computation of transcendental functions and pixel/vertex manipulations. The MT-Issue unit is in charge of mapping active threads on the available SPs.

A multiprocessor is able to concurrently execute groups of 32 threads called *warps*. Since each thread in a warp has its own control flow, their execution paths may diverge due to the independent evaluation of conditional statements. In this case, the warp serially executes each path. When the warp is executing a

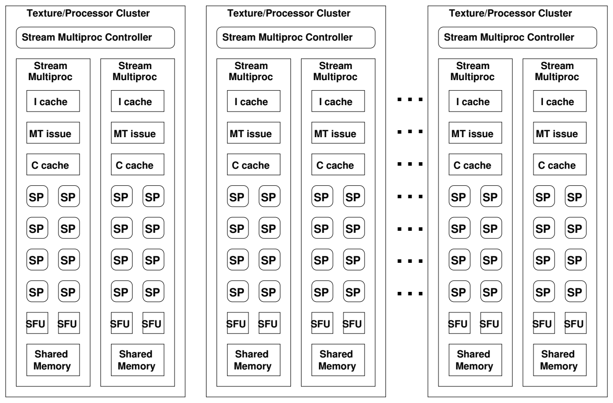

Figure 1. Sketch of the NVIDIA G80 streaming processors array architecture: each Texture/Processor Cluster contains two stream multiprocessors. In turn, each stream multiprocessor is composed of eight streaming processor cores (SP), plus two special function units (SFU). Shared memory is local to each stream multiprocessor.

given path, all threads that have not taken that path are disabled. On the other hand, in case the control flows converge again, the warp may return to a single, parallel execution of all threads. Each multiprocessor executes warps in a fashion much like the *Single Instruction Multiple Data* (SIMD) paradigm, since every thread will be assigned to a different SP and every active thread will execute the same instruction on different data.

The MT-Issue unit weaves threads into a number of warps and schedules an active warp for execution, using a round-robin scheduling policy with aging for this purpose.

Streaming multiprocessors are in turn grouped in Texture Processor Clusters (TPC). Each TPC includes two streaming multiprocessors in the G80 architecture. The TPC also includes support for Texture processing, though these features are seldom used for general purpose computing and will not be investigated in this paper.

Finally, the NVIDIA GPU on-board memory hierarchy includes registers (private to each SP), on-chip memory and off-chip memory. The on-chip memory is private to each multiprocessor, and is split into a very small instruction cache, a read-only data cache, and 16 KB of addressable shared data, respectively indicated as I-cache, C-cache and Shared Memory in Figure 1. This shared memory is organized in 16 banks that can be concurrently accessed, each bank having a single read/write port.

3.2. CUDA Programming Model

The Compute Unified Device Architecture (CUDA) [20], [18], proposed by NVIDIA for its G80, G92 and GT200 graphics processors, exposes a programming model that integrates host and GPU code in the same C++ source files. The main programming structure supporting parallelism is an explicitly parallel function invocation (*kernel*) which is supposed to be executed by a user-specified number of threads. Every kernel is explicitly invoked by host code and executed by the device, while the host-side code continues execution asynchronously after instantiating the kernel. The programmer is provided with a specific synchronizing function call to wait for the completion of the active asynchronous kernel computation.

The CUDA programming model abstracts the actual parallelism implemented by the hardware architecture, providing the concepts of *block* and *thread* to express concurrency in algorithms. A block captures the notion of a group of concurrent threads. Blocks are required to execute independently, so that it has to be possible to execute them in any order (in parallel or in sequence). Therefore, the synchronization primitives semantically act only among threads belonging to the same block. Intra-block communications among threads use the *logical shared memory* associated with that block.

Since the architecture does not provide support for the message-passing techniques, threads belonging to different blocks must communicate through *global memory*. The global memory is entirely mapped to the off-chip memory. The concurrent accesses to logical shared memory by threads executing within the same block are supported through an explicit barrier synchronization primitive.

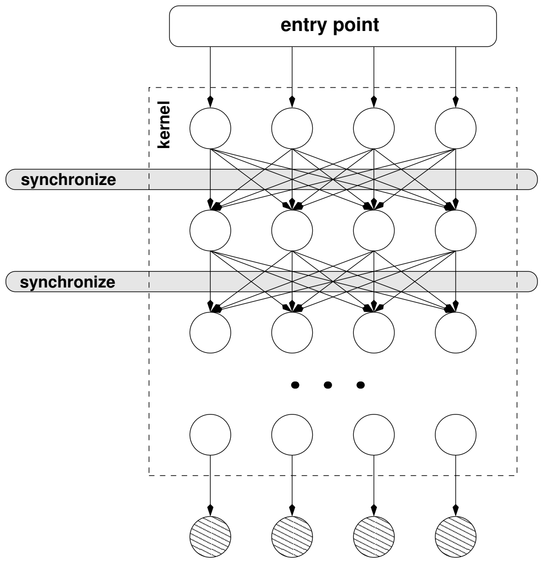

Figure 2. AES Dataflow Graph: each empty node represents an operation that manipulates a single word of the AES state matrix, while a shaded node represents the copy of an output value to the global memory. An operation row implements a round of the algorithm. Synchronization between each round and the next is required. These synchronizations are explicit in fine-grained implementation where each thread is responsible for the execution of a column of operations. In the coarse-grained implementation, a single thread executes the whole kernel, thus removing all explicit synchronizations.

A kernel call-site must specify the number of blocks as well as the number of threads within each block when executing the kernel code. The current CUDA programming model imposes a capping of 512 threads per block.

The mapping of threads to processors and of blocks to multiprocessors is mainly handled by hardware controller components. Two or more blocks may share the same multiprocessor through mechanisms that allow fast context switching depending on the computational resources used by threads and on the constraints of the hardware architecture. The number of concurrent blocks managed by a single multiprocessor is currently limited to 8.

In addition to the logical shared memory and the global memory, in the CUDA programming model each thread may access a *constant* memory. An access to this read-only memory space is faster than one to global memory, provided that there is sufficient access locality since constant memory is implemented as a region of global memory fit with an on-chip cache. Finally, another portion of the off-chip memory may be allocated as a *local memory* that is used as thread private resource. Since the local memory access is slow, the shared memory also serves as an explicitly managed cache – though it is up to the programmer to warrant that the local data being saved in shared memory are not accessed by other threads. Shared memory comes in limited amounts (threads within each block typically share 16 KB of memory) hence, it is crucial for performance that each thread handle only small chunks of data.

3.3. Design of a Parallel AES

The design of a parallel implementation of the AES algorithm presented in Section 2 is chiefly dependent on the choice of the grain of parallelism to expose.

We define *fine-grained design* a solution exposing the internal parallelism of each AES round. The four 32-bit words of the state computed by each AES round can be manipulated operating independently one from each other. As shown in Figure 2, this strategy explicitly points out a way to concurrent compute the state of each AES round. The computation of each state word needs to read the value of the former round state, thus forcing the insertion of a barrier synchronization between consecutive rounds.

On the other hand, we define *coarse-grained design* a solution that ignores the internal parallelism of the algorithm, and focuses instead on the higher-level parallelism exposed by operation modes such as the ECB and CTR. In these schemes, different blocks can be encrypted or decrypted by different instances of the algorithm running in parallel. Our implementation will tackle the CTR mode, since it is more secure than ECB, as discussed in Section 2. This design choice is also strongly dependent on the architecture of the GPU – the level of hardware parallelism affects the tradeoff point between coarse- and fine-grained designs, as will be shown in Section 4.

A second design choice is the storage location of the four T-boxes, T_0, T_1, T_2 and T_3. Since they contain exclusively read-only data, they may be loaded into the constant memory. However, this may not be the optimal choice, since the benefits of the constant memory depend closely on the locality of the accesses. Since the design criteria of the substitution boxes T_i is to be accessed in a way that is dependent on the input values and therefore not predictable, the accesses locality is practically non-existent.

An alternative choice is to load the T-boxes in shared memory. Shared memory accesses are much faster than non-cached constant memory accesses, since the memory is located within each stream multiprocessor. On the other hand, shared memory typically experiences a heavy load, being shared by 8 processing units and limited in size to slightly less than 16 KB storage. In the AES case, though, these issues are not crucial, since the algorithm works on a small data set: four 32-bit words representing the state, plus the T-boxes, which are shared by all threads within a block. In the fine-grained implementation, four threads cooperate to compute a single AES block. Since there is a need to access the values computed in the previous round, two sets of four 32-bit words are used to keep the previous state values while computing the new ones, and are shared by the four threads. The coarse-grained implementation, on the other hand, uses only a single set of four 32-bit words per thread. Therefore, in the fine-grained implementation, each shared

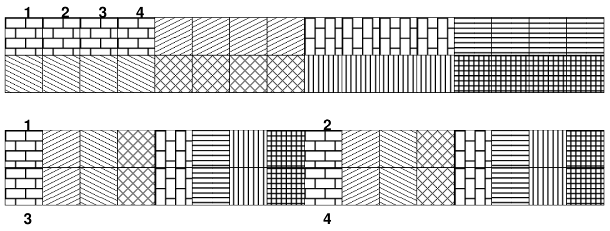

Figure 3. Use of banks of shared memory: banks are represented by columns of data blocks, while numbers mark the memory access sequence of the thread of index 0. In the upper part of the figure, the shared memory is partitioned naively, leading to 16 bank access collisions among 16 threads, while in the lower part the shared memory is partitioned in order to avoid any collisions.

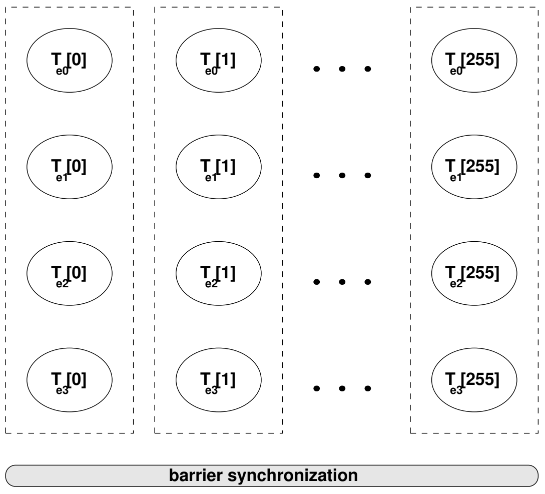

Figure 4. T-boxes loading on shared memory: the process is fully distributed over 256 threads. Since the shared memory is private to each stream multiprocessor, this operation is performed in parallel on all of them.

memory is used to hold $8 \cdot n_{threads}$ bytes of state, and 4 KB of T-boxes, while in the coarse-grained implementation the shared memory holds $32 \cdot n_{threads}$ bytes of state, plus the same amount of space for the T-boxes. Therefore, in fine-grained implementation $n_{threads}$ may scale up to 512 (the maximum value allowed by the architecture, leading to the occupation of 8 KB of shared memory), while in coarse-grained implementations, it is capped to 256 (with a shared memory occupation of 12 KB).

The computation of each AES round needs to access the four T-boxes in sequential order. To avoid bank access collisions, it is important to allocate the memory used by each thread in a way that as few threads as possible use the same banks at the same time.

Figure 3 provides a visual representation of the collisions occurring during the access to shared memory banks. A naive allocation of the memory causes the i-th thread to access the memory addresses from $4i$ to $4i + 3$. Since there are 16 memory banks, threads i and $i + 4$ collide on every access. A better allocation makes the i-th thread access memory at locations i, $i + n_{threads}$, $i + 2n_{threads}$, $i + 3n_{threads}$, thus avoiding collisions.

Finally, the T-boxes will need to be loaded into the shared memory of each stream multiprocessor, thus requiring an initialization step. As shown in Figure 4, the loading process can be spread over 256 threads per stream multiprocessor, thus minimizing the time spent in the initialization step.

4. Experimental Results

The experimental campaign reported in this section aims at assessing the feasibility of employing GPUs as cryptographic coprocessors, as well as identifying the most effective parameter choice for the proposed parallel implementation of the AES block cipher. In order to make a sound choice we provide comparisons between the GPU and CPU based implementations of AES, as well as a discussion of the different implementations (coarse-grained vs. fine-grained, shared memory vs. constant memory), showing their efficiency on different input plaintext sizes.

4.1. Experimental Settings

For our experiments, we have used two NVIDIA graphics boards of the GeForce 8 family [7]: an 8400 GS with a single Texture/Processor Cluster (and therefore 16 stream processors), and an 8800 GT with seven Texture/Processor Clusters (112 stream processors). Thus, our experimental campaign covers the current range of low-cost graphics hardware, ranging from 30$ to 100$. We implemented the four proposed approaches to AES CTR-mode encryption within the OpenSSL framework.

As for the plaintext sizes, we report results in the range from 4 KB to 128 MB. This range was chosen for the purpose of comparing our results with previous works. However, in practical applications such as securing web services, the plaintext size typically ranges from 35 KB to 150 KB [6], so particular attention must be focused on the performance of encryption algorithms when used on this input size range.

A further key application of cryptographical accelerators is lowering the latency required to access a fully encrypted hard-disk volume. Typical chunk sizes (a chunk being the disk-volume unit of transfer) are in the same range mentioned above, so also the encryption of large data on a disk or multi-disk storage (RAID) may largely benefit from the use of a low cost graphic hardware used as cryptographic accelerator.

The parameter space considered for the AES algorithm implementation in the experimental campaign is summed up as follows:

- *Kind of parallelism*: either fine-grained or coarse-grained.
- *T-box memory allocation*: either shared memory or constant memory.
- *Input plaintext size*: ranging from 4 KB to 256 MB. The input size implicitly defines the number of threads that will be launched.
- *Number of CUDA blocks and CUDA threads*: threads are packed in blocks of sizes ranging from 8 to 256 for the coarse-grained design and up to 512 for the fine-grained one.

4.2. Performance Evaluation

We report two different types of results. First of all, we show the throughput rate as a function of the input plaintext size, considering the optimal configuration of the CUDA blocks and threads arrangement. Afterwards, we show the impact of grouping threads into blocks of variable size, while keeping the plaintext size fixed.

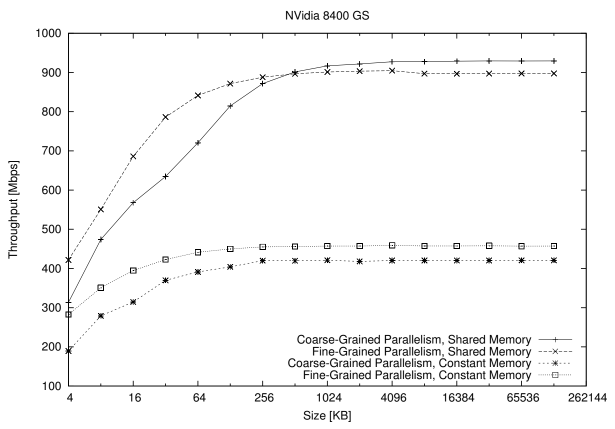

Figure 5. Throughput for the four AES implementations on NVIDIA 8400 GS (16 SP).

Figures 5 and 6 show the results of the experimental campaign on the 8400 GS and 8800 GT boards, respectively. In each figure, the AES ciphertext throughput is charted as a function of the size of the input plaintext. It can be seen that the implementations using constant memory fail to provide good performance, confirming the soundness of using shared memory to keep the T-boxes in. It is also possible to see that the fine-grained approach outperforms the coarse-grained one in the 35 KB to 150 KB plaintext size range that was identified as crucial for practical applications. The break-even point is reached at about 512- KB on the 8400 GS and 2 MB on the 8800 GT, respectively; further pointing out the relevance of the fine-grained approach in order to fully exploit the massive number of stream processors even with a small sized input.

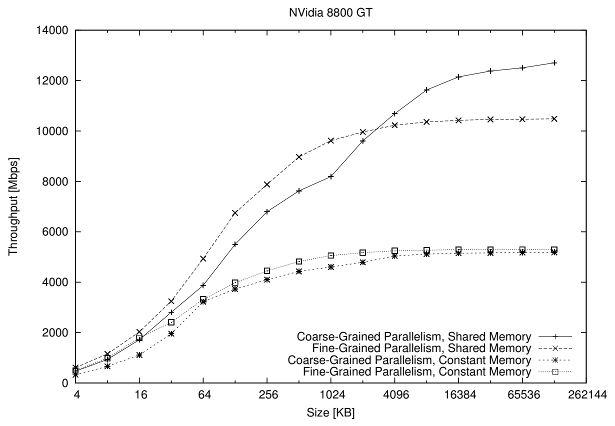

Figure 6. Throughput for the four AES implementations on NVIDIA 8800 GT (112 SP).

Before the break-even point the fine-grained approach proves more effective mostly because it provides a wider computational load thanks to the four times higher number of threads spawned with respect to its the coarse-grained counterpart when considering a fixed input size.

With the increase of the input size to architectural resources ratio the coarse-grained approach has the upper hand on the fine-grained as far as performances go. Such behavior has to be ascribed to the fact that now both designs equally spawn enough threads to flood the architecture with computational load. Moreover, the coarse-grained approach gets an higher IPC since in this case the spawned threads do not need any form of synchronization.

Consequentially, when considering an hardware architecture with more computational resources such as the 8800GT w.r.t. the 8400GS, the break-even point between the two design approaches shifts toward higher input sizes since the input size to architectural resources ratio is lower.

Finally, the asymptotic trend depicted by Figures 5 and 6 correctly exhibits the fact that an increasing number of threads without scaling the hardware resources accordingly, forces the massive marshalling of the spawned threads (explicit serialization).

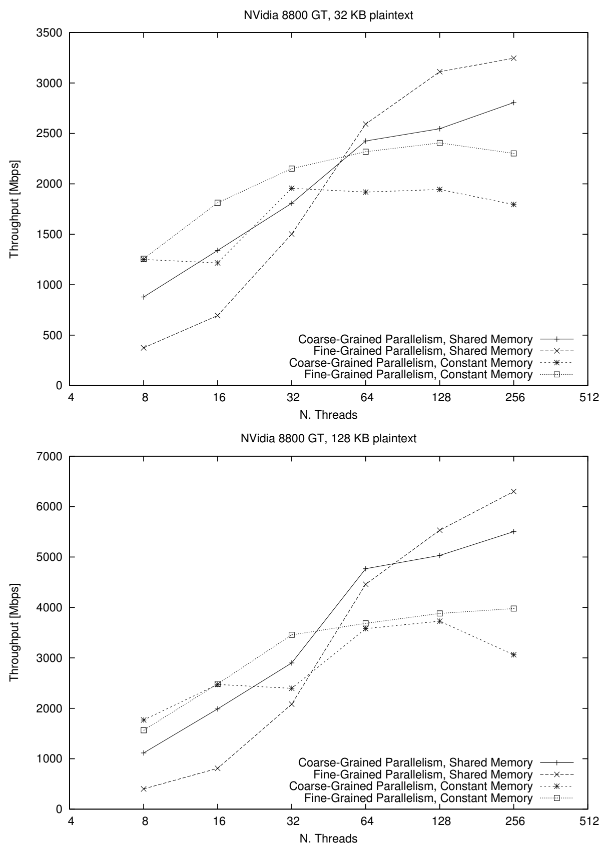

Figure 7. Impact of the number of threads per block on the AES throughput; the cases of 32KB and 128KB plaintext size are shown for the NVIDIA 8800 GT (112 SP).

Figure 7 depicts the impact of the number of CUDA-threads assigned to each CUDA-block on the throughput rates. We report for this experiment results from the larger 8800 GT board and two plaintext sizes, 32 KB and 128 KB, that represent well the expected working conditions of the ModSSL encryption module of an Apache web server.

The programmability features exposed by the CUDA framework offer the potential to obtain the best performance when the hardware-resources are fully used at their best. To this purpose, the number of CUDA-threads per CUDA-block and the number of CUDA-blocks actually instantiated by an algorithm implementation has to be exhaustively explored to find the best values suited for a specific input size.

Threads from the same CUDA-block are mapped to the multicore architecture by the MT-issue unit, which splits them into groups of at most 32 threads each (*warp*), and issues the instructions at warp level on a single streaming multi-processor. Warp instructions coming from different warps of the same block may be interleaved one with each other in order to hide latencies intrinsic to some instructions.

Figure 7 shows clearly how CUDA-blocks containing less than 32 threads obtain a pitiful performance due to them under-filling every warp, and thus not fully exploiting the parallelism offered by the hardware. The use of a multiple of 32 threads per CUDA-blocks (and thus allowing each block to be composed of several warps) reveals a sensible growth in performance since full warps are seamlessly interleaved hiding memory access latencies. When moving from 32 threads per CUDA-block to 64, there is little cost in terms of bank conflicts, thus the performance improvement is high: the scheduling queue in the MT-issue unit is able to hold up to 24 warps, but choosing them from 8 blocks only – thus the number of schedulable warps doubles in this case. As the number of threads per CUDA-block grows, the number of different blocks that can be in the scheduling queue at the same time decreases (since each block will be composed of more than 3 warps), thus limiting the benefits obtained, since the ability of the scheduler to mask synchronization delays by scheduling warps from different blocks decreases.

Note that, in the case of the fine-grained implementation, we use four times the number of CUDA-blocks than in the corresponding coarse-grain implementation with the same number of threads per CUDA-block – thus, for this implementation, the scheduler finds it easier to balance the workload across the different multiprocessors, leading to a performance benefit. This also explains why the fine-grained implementation obtains worse performances when using only 8 or 16 threads per CUDA-block: it produces a large number of very small CUDA-blocks, which will not be ready in the scheduling queue at the same time due to the architectural constraint mentioned above. The coarse-grain implementation, on the other hand, produces less CUDA-blocks, thus allowing a better schedule.

In the case of the implementations using constant memory, there is a single read access port, so the number of colliding bank accesses roughly doubles at every doubling of the threads number. Thus, the advantage obtained from the augmented number of in-flight threads is dominated by the additional memory access latencies. The throughput growth trend of the constant memory solutions, remains within a smaller range of improvement (only a twofold increase against a 32x increase in the thread number).

Finally, the performance trend exhibited by the shared memory solutions benefit in a more significative way from the increasing number of threads because the shared memory has 16 read/write access ports and a significantly lower constant access latency. The number of threads varies between 8 and 256 the throughput improvement ranges from 8x with 32 KB of input plaintext to 16x with 128 KB of input plaintext. In particular with 128 threads or more the fine-grained approach yield better results.

In order to give a reference for the performance figures obtained from the GPUs we provide in Table 1 some benchmark data extracted from four common CPUs. We chose Intel's Core 2 Quad Q6600 and Xeon Clovertown E5335 as two current high-end processors and both AMD's Athlon 64x2 3800+ and Intel's Pentium D 540 as two low-end but widely deployed ones. As far as the operating system goes, we chose to run the benchmarks

Table 1. Performance comparison: Throughput [Mbps]

Plaintext Size	NVIDIA 8800 GT	NVIDIA 8400 GS	Intel Core 2 Quad	AMD Athlon 64x2	Intel Xeon E5335	Intel Pentium D 540
32 KB	2917	771	862	207	721	531
128 KB	6591	855	868	207	724	534
32 MB	12075	908	857	172	709	498
128 MB	12412	909	856	170	708	493
Processor Specifications	112 SP	16 SP	64-bit mode	32-bit mode	64-bit mode	32-bit mode
Clock Freq. [MHz]	1500	900	2400	2000	2000	3200
Cost [$]	∼170	∼30	∼180	∼45	∼900	∼40

on Linux, in full 64 bit mode for the two newer processor,since it's definitely likely that they will be running newly developed software. The two older units were benchmarked using a 32 bit version of Linux, since, representing old deployed systems, it is more likely that the will not be running a 64 bit distribution. A relevant remark to be made is that the OpenSSL toolkit provides an assembly level optimized version of AES for the x86-64 architectures, thus yielding significantly better results, although it is only available for full 64 bit distributions.

As it can be seen from the reported results the GPU based implementations are almost always faster than the CPU implementations, with the lone exception of the 8400 GS underperforming w.r.t. Intel's Core 2 Quad when dealing with small plaintext sizes. This may be ascribed to the threefold difference in clock rates between the two processing units and the large 4 MB on-die cache of the latter unit which allow the CPU to perform faster (albeit only by a small margin) than the GPU.

The best throughput results are obtained with the higher-end NVIDIA 8800 GT that, through scaling up the plaintext size, is able to achieve a peak performance of 14 times the throughput of the fastest CPU (12.4 Gb/s), and keeps yielding between 3 and 7 times improvements when employed in the 32 to 128 KB range that it is considered the most relevant for applicative purposes.

When comparing the performance of the two NVIDIA GPUs, we expect, from the difference in the number of parallel processors, a 7-fold improvement in favor of the 8800 GT. This improvement is fully obtained only for large input plaintext sizes, which provide enough computational load to fill the processing capabilities of the more powerful GPU, which actually proves even faster than expected, achieving a throughput improvement of more than 13 times with respect to the 8400 GS, thanks to a higher clock rate.

The results clearly point out an advantage in using GPUs to compute cryptographic primitives, since, not only they are able to fulfill the throughput needs far better than general purpose CPUs do, but they also free up valuable cpu time which can be spent in dealing with higher OS or application loads.

When evaluating performances with respect to hardware costs, the two proposed boards are able to offer a better tradeoff in terms of throughput per dollar than the classical solutions, and they are easily integrable with existing machines, without issue when running in both 32 and 64 bit mode. In particular the NVIDIA 8400 GS offers performances comparable to a Core 2 Quad at a sixth of the price, while the 8800 GT offers 9 times the encryption speed w.r.t. the Xeon E5335 at only a fifth of the price, bringing the combined cost-performance advantage up to 45x. All the price estimates for the CPUs are actually done by defect, since, while adding a graphic card to a running server can be done without upgrading other hardware, the addition of a second CPU is not always possibile in such an effortless way, therefore the effective cost-performance advantages are even stronger than the one we assumed in our conservative hypothesis.

There are also hardware solutions, such as the VIA PadLock [21], to perform AES. The VIA chip is reported to provide a top throughput of 1.9 GB/s, which remains below the throughputs provided by graphics hardware such as 8800 GT.

5. Related Works

Manavski [22] pioneered the use of the CUDA framework to implement AES – though it does not specify which mode of operation is implemented. However, the implementation proposed in [22] does not efficiently exploit the hardware parallelism offered by the NVIDIA 8800 GTX used in its experimental evaluation: when compared to our or other recent works using similar boards, it only reaches half of the throughput rates reported in these works. The reason of the reduced performances of the AES implementation [22] is in the use of constant memory for the T-boxes. As we show in Section 4, the AES designs using constant memory never reach the performance of shared memory-based designs.

Harrison and Waldron [17] first proposed a study of AES implementation on GPU hardware, using the GeForce 6 and 7 families. These GPUs are not based on the G80 processor architecture, but rather on traditional vector processors, not supporting integer operations or a general purpose programming interface such as CUDA. Indeed, the AES implementations proposed in this study are based on the OpenGL library, which is not geared towards general purpose computing. Even with these limitations, their study successfully demonstrates the effectiveness of the GPU as a coprocessor for bulk data encryption and decryption.

A more recent work by Harrison and Waldron [23] proposes a CUDA-based implementation of AES, comparing it to previous CPU-based and OpenGL-based implementations. Their approach is comparable with the coarse-grained, shared memory design discussed in Section 3. Section 4 reports experimental measures consistent with those in [23] – the differences between the figures we report for the coarse-grained shared memory design are due to the different hardware platforms: the nearest comparison is between the NVIDIA 8800 GT, which has 112 streaming processors, used in our experiments and the 128 processors NVIDIA 8800 GTX used in [23]. Moreover, the 384-bit memory interface of the board used by [23], compared with our 256-bit memory interface is also a significant hardware-specific advantage. With respect to [23], we offer the exploration of different design solutions, including the exploitation of fine-grained parallelism, which proves more effective under a wide range of plaintext input sizes, in particular those expected to fit within the most typical range for web applications.

6. Conclusions

In this paper, we investigate the possibility of using modern graphics hardware, supported by toolchains oriented to general purpose programming, as a coprocessor to ease the CPU load when encrypting/decrypting data streams in web server applications. We have shown how to effectively implement the AES block cipher using the CUDA toolchain and programming model, extracting as much parallelism as possible from the algorithm with both coarse and fine grained approaches. We provided an extensive quantitative evaluation on a range of NVIDIA GPUs based on the G80 architecture and scaling from 16 to 112 cores. These experiments allow us to confirm that, for the AES block cipher and similar algorithms, it is possible to efficiently use the GPU as a coprocessor. Moreover, this solution is cost effective when compared to the assembly-level optimized CPU-based implementations of the AES built in the OpenSSL library.

With respect to previous works in the field, we provided an experimental evaluation of a much larger design space, showing two locally optimal solutions with respect to the input plaintext size parameter. These solutions, based on fine- and coarse-grained parallel designs, may be integrated to provide better performance over the entire range of the input parameter.

Overall, we report throughput improvements of up to 14 times over the CPU implementations chosen as baseline, as well as performance/cost figure of about 73 Mbps per dollar for the NVIDIA 8800 GT against the 4 Mbps per dollar of the Intel Core 2 Duo.

Future research efforts may address the automated exploration of the design space by the compilation toolchain, a task that is bound to require a major effort from the designer with the current tools, as well as the deployment of automatically-tuned solutions specific to input size. On a different direction, the use of the GPU as a complete, low-cost cryptographic coprocessor could be further explored through the parallel-oriented re-engineering of other cryptographic primitives integrated in widespread tools such as OpenSSL.

References

[1] Apache Foundation, “Apache HTTP Server Project,” http://httpd.apache.org, Oct. 2008.

[2] OpenSSL Project, “OpenSSL,” http://www.openssl.org.

[3] Netcraft LTD., “July 2008 Web Server Survey,” http://news.netcraft.com/archives/web_server_survey.html, Jul. 2008.

[4] IETF, “The tls protocol version 1.0,” http://www.ietf.org/rfc/rfc2246.txt, January 1999.

[5] Ralf S. Engelschall and Ben Laurie, “The Apache interface to OpenSSL,” http://www.modssl.org, Oct. 2008.

[6] R. Levering and M. Cutler, “The portrait of a common html web page,” in *DocEng ’06: Proceedings of the 2006 ACM symposium on Document engineering*. New York, NY, USA: ACM, 2006, pp. 198–204.

[7] NVIDIA Corporation, “GeForce Graphics Processors,” http://www.nvidia.com/object/geforce_family.html, Sep. 2008.

[8] National Institute of Standards and Technology (NIST), “FIPS-197: Advanced Encryption Standard,” http://www.itl.nist.gov/fipspubs/, Nov. 2001.

[9] ——, “FIPS-46-3: Data Encryption Standard (DES),” http://www.itl.nist.gov/fipspubs/, May 1999.

[10] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[11] M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Methods and Techniques,” National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-38a, 2001, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[12] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, “A concrete security treatment of symmetric encryption,” in *FOCS ’97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS ’97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 394.

[13] H. Lipmaa, P. Rogaway, and D. Wagner, “CTR-mode encryption,” in *First NIST Workshop on Modes of Operation*, 2000. [Online]. Available: citeseer.ist.psu.edu/article/lipmaa00ctrmode.html

[14] N. Modadugu and E. Rescorla, “AES Counter Mode Cipher Suites for TLS and DTLS,” Internet Engineering Task Force (IETF) Internet-Draft draft-ietf-tls-ctr-01, Jun. 2006.

[15] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[16] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, “CryptoGraphics: Secret Key Cryptography Using Graphics Cards,” in *CT-RSA*, ser. Lecture Notes in Computer Science, A. Menezes, Ed., vol. 3376. Springer, 2005, pp. 334–350.

[17] O. Harrison and J. Waldron, “AES Encryption Implementation and Analysis on Commodity Graphics Processing Units,” in *CHES*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds., vol. 4727. Springer, 2007, pp. 209–226.

[18] NVIDIA Corporation, “CUDA Technology,” http://www.nvidia.com/CUDA, Sep. 2008.

[19] T. R. Halfhill, “Parallel Processing With CUDA,” *Microprocessor Report*, Jan. 2008.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[21] VIA Technologies, Inc., “VIA PadLock Security Engine,” http://www.via.com.tw/en/initiatives/padlock/, Aug 2005.

[22] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” in *IEEE International Conference on Signal Processing and Communication, ICSPC 2007*, Nov. 2007, pp. 65–68.

[23] O. Harrison and J. Waldron, “Practical Symmetric Key Cryptography on Modern Graphics Hardware,” in *17th USENIX Security Symposium*. USENIX, Jul. 2008, pp. 195–209.