Fast Disk Encryption Through GPGPU Acceleration

Giovanni Agosta, Alessandro Barenghi, Fabrizio De Santis, Andrea Di Biagio
Politecnico di Milano
{agosta,barenghi,dibiagio}@elet.polimi.it, fabrizio.desantis@mail.polimi.it

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

***Abstract*—We present the design and performance analysis of a GPU-optimized implementation of a disk encryption application employing the XTS mode of operation applied together with the Twofish algorithm within the well-known TrueCrypt suite. We show how to correctly tune the design parameters, including data allocation, thread packing, and parallelization strategy. Overall, our implementation of TrueCrypt running on a NVidia GTX260 GPU outperforms by 67% the baseline implementation running on a four core CPU.**

I. Introduction

Data storage encryption has always been a key point in warranting confidentiality, but encrypting large disk volumes imposes a significant computational load on the CPUs. In case of a single host system the CPU time used for encryption is subtracted from the one available for coping with the user's needs. On dedicated Network Area Storage managers (NAS) throughput is sacrificed in order to deal with the additional workload due to encryption. A way to reduce this computational load is to employ dedicated ASIC coprocessors but this comes at a major cost, especially on user machines. Moreover, the accelerators must be designed according to each platform specification and are typically customized for a single encryption algorithm with fixed key sizes [11], thus resulting in a not so flexible solution.

A viable alternative comes from the Graphics Processing Unit (GPU) world where coprocessors have grown 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 without the need of fitting through the graphics specific interface, they may be a viable choice for the implementation of generic computationally demanding algorithms. The advantage of such a choice lies in both using off-the-shelf hardware which comes at a very low cost with respect to ASIC solutions, and exploiting the already deployed installbase on common personal computers. Moreover, this design solution relies on a standard bus interface (PCI-Express) and on a subset of the recently standardized OpenCL language [12] for high performance computing on graphic hardware.

The use of GPUs to speed up the computation of cryptographic primitives was pioneered by D. Cook *et al.* in [6]. Further developements [9] focused on the engineering of an AES implementation oriented towards the use in SSL network communication channels, and considered the ECB, CBC and CTR modes of operation.

The goal of this paper is to analyse the design and performance of a GPU-optimized implementation of a disk encryption application employing the XTS mode of operation applied together with the Twofish algorithm [17] within the well-known TrueCrypt suite [1]. The XTS mode of operation has been recognized as the standard for disk encryption, and Twofish is commonly used for this application, though the results of our analysis apply with minimal modifications to other encryption algorithms when used in the same context. The target GPU architecture for this study is NVidia GeForce GT200 family.

The rest of this paper is organized as follows. Section II describes the XTS mode of the operation and its use within the TrueCrypt disk encryption software. Section III describes the GeForce GT200 GPU family and its programming model. Section IV describes the design of an optimized GPU implementation of TrueCrypt, relative to the Twofish algorithm. Section V provides an experimental evaluation of the TrueCrypt implementation on the NVidia GT200 architectural family. Finally, Section VI provide some conclusions and points out some directions for future work.

II. TrueCrypt XTS Encryption Algorithm

The encryption of massive amounts of data on the fly is one of the most computationally intensive operation performed by both storage servers and common single user systems in order to to enforce confidentiality. Among all the applicative solutions only a few are able to operate through the XTS mode of operation [16], which was recognized as the IEEE P1619 standard for data encryption on block-oriented storage devices [2], [3], [10]. Within those compliant to IEEE P1619, TrueCrypt is a multi-platform and open source software that manages virtual disks to be encrypted on the fly. The software is able to use either files, disk partitions or full devices as data support for the logical volumes (a.k.a. virtual disks), thus allowing its employment on both single disks and multiple network shared storages.

The XTS mode of operation was designed primarily for the encryption of data on block oriented devices and therefore assumes that the plaintext is naturally split into *data units*. The underlying assumption motivating the aforementioned organization of data is the inherent structure of permanent storage devices which allows the access to data only in physical blocks ranging up from 512 bytes. These data units are usually larger than the width of the block cipher employed in the process, and must therefore be further split into smaller *cipher blocks*,

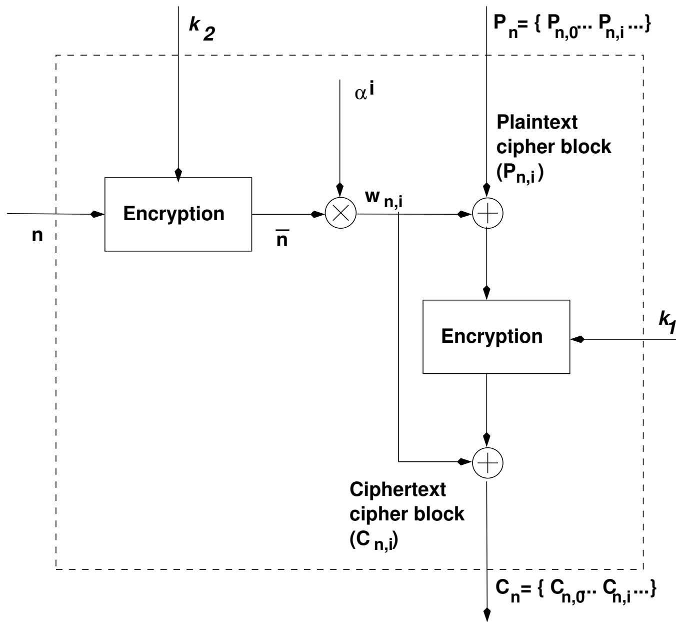

Fig. 1. XTS Encryption Mode block diagram [3]

which must have a size that fits exactly the input data size of the encryption algorithm.

The tweakable codebook mode of operation XTS, shown in Figure 1, exploits the Xor-Encrypt-Xor design [16] to obtain resilience against the wider range of correlation and distinguishing attacks [8] made feasible by the large amount of encrypted data available to the attacker.

The input of the encryption mode is represented by both the plaintext P, split into data units P_n, and the positional indices n of the data units within the logical volume. The enciphered data unit C_n is obtained through three steps, which act block-wise on the input data. At first, the input block $P_{n,i}$ is whitened through xor-ing with an appropriate position-dependent pad $w_{n,i}$. Then, the result is encrypted using the chosen symmetric encryption algorithm with the primary encryption key k_1. After the encryption, the output is whitened again with the same pad $w_{n,i}$ to obtain $C_{n,i}$.

As depicted in Figure 1, the whitening pad $w_{n,i}$ is obtained in two steps. At first, the data unit index n is enciphered with a secondary key k_2. The output of the encryption $\bar{n}$ is interpreted as an element of $\mathbb{Z}_{2^{128}} \simeq \mathbb{Z}_2(\alpha)$, where:

$$\mathbb{Z}_2(\alpha) = \{\beta | \beta = \theta_0 \alpha^{127} + \ldots + \theta_j \alpha^{127-j} + \ldots + \theta_{127}\}$$

with $\forall j,\ j \in \{0, \ldots, 127\}\ \theta_j \in \mathbb{Z}_2$ and $\alpha \in \mathbb{Z}_{2^{128}} \backslash \mathbb{Z}_2$ is the primitive element of $\mathbb{Z}_{2^{128}}$ such that $\alpha^{128} + \alpha^7 + \alpha^2 + \alpha + 1 = 0$ with binary representation $\alpha = (00\ldots010)_2$. This value is used to obtain the whitening pads $w_{n,i}$ for all cipher blocks $P_{n,i}$ within a data unit P_n, through multiplying it in $\mathbb{Z}_{2^{128}}$ by α^i, where i is the sequence number of cipher block $P_{n,i}$ within P_n.

The TrueCrypt framework limits the number of data units enciphered with a single call to the encryption primitive to 512, that is, every call encrypts a chunk of 256 KB from the plaintext. While this is not an explicit constraint imposed by the standard, it has multiple advantages. First of all, it represents a tradeoff between the dynamic memory footprint and the performance gain from enhanced data locality of the memory segment. Moreover, since the ciphertext will be written directly onto the physical storage device, and the Operating System clusters consecutive write operations, it is sensible [19] to choose the plaintext chunk passed to the encryption primitive as a multiple of the largest possible physical block size (starting from 512 bytes up). Since the actual physical device may be composed of an array of redundant disks (RAID), the physical block size may ramp up to 128 KB. On the other hand, a physical device composed by a single disk has a physical block size of 4 KB. A 256 KB sized plaintext chunk represents a well-known best practice trade-off between underfilled calls to the encryption primitive and the risk of handling a single physical block encryption through multiple calls, thus harming disk writing throughput.

Truecrypt offers the possibility to choose the encryption algorithm E among the three best candidates of the NIST AES Contest: AES [7], Twofish [17] and Serpent [4]. All the algorithms are employed with 256 bit wide keys in order to compensate the potential risk of correlation attacks which could be lead owing to the large quantity of enciphered material. The size of the input to all the available encryption algorithms is 128 bits, therefore resulting in 32 cipher blocks per data unit. In this paper, we chose Twofish as the encryption algorithm, since it has the advantage of being faster than AES when used with 256 bit wide keys [18] while retaining the same, very high, security margin.

Algorithm II.1 summarizes the XTS mode of operation implemented by the TrueCrypt toolkit.

III. Programmable Graphics Processing Units

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. Pioneering works attempted to adapt “general purpose” applications using graphic rendering APIs (OpenGL and DirectX) since they were the only way to tap into the GPU computational resources [15].

A. The NVIDIA GT200 Architectures

Modern GPUs now include hundreds of processing elements grouped in a hierarchical structure. In our case, the NVIDIA GT200 GPU series provides a set of independent multithreaded streaming multiprocessors. Figure 2 shows an overview of the NVIDIA GT200 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 8 streaming processors, two special functional units and a multithreaded instruction issue unit (respectively indicated as SP, SFU and MT-Issue in Figure 2). A SP is a fully pipelined single-issue core with two ALUs and a single floating point unit (FPU). SFUs are dedicated to

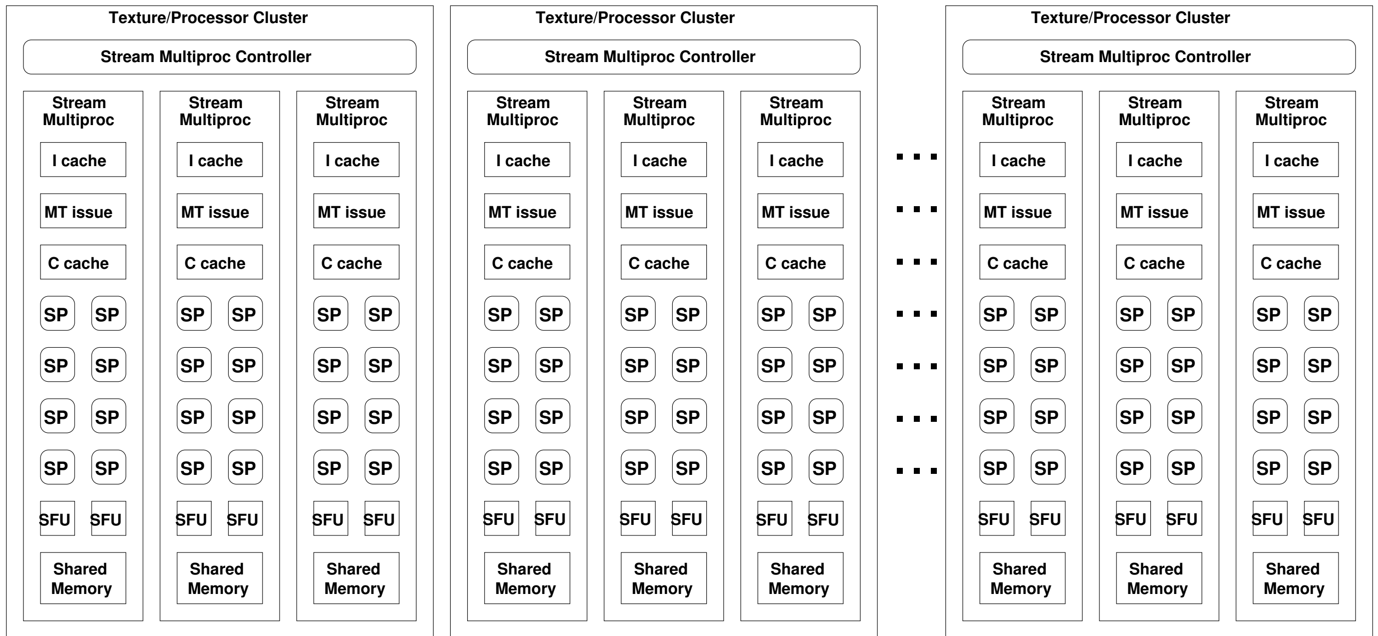

Fig. 2. Sketch of the NVIDIA GT200 streaming processors array architecture: each Texture/Processor Cluster contains three stream multiprocessors. In turn, each stream multiprocessor is composed of eight streaming processor cores (SP), plus two special function units (SFU). Shared memory is local to each stream multiprocessor.

Algorithm II.1: TrueCrypt XTS Encryption Mode Algorithm.

Input: $P = (P_d, P_{d+1}, \ldots, P_{d+t})$: input plaintext chunk as a sequence of t data units. [d, d+t]: range of data unit indices of the whole plaintext to be encrypted through multiple calls to the algorithm. k_1: primary encryption key. k_2: secondary key.
Output: $C = (C_d, C_{d+1}, \ldots, C_{d+t})$ output ciphertext chunk.
Data: $E(p, k)$: symmetric encryption function on plaintext p using key k. $t = 512$: number of data units in a plaintext chunk.
$m = 32$: number of cipher blocks per data unit.
$P_n = (P_{n,0}, \ldots, P_{n,m-1}), C_n = (C_{n,0}, \ldots, C_{n,m-1}) \; \forall \; n \in [d, d+t]$.

1 **begin**
2   **for** $n \leftarrow d$ **to** $d + t$ **do**
3         $\bar{n} \leftarrow E(n,\ k_2)$
4   **for** $i \leftarrow 0$ **to** $m - 1$ **do**
5             $w_{n,i} \leftarrow \bar{n} \otimes \alpha^i$
6             $C_{n,i} \leftarrow E(w_{n,i} \oplus P_{n,i},\ k_1) \oplus w_{n,i}$
7   **return** C
8 **end**

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; when this happens, the warp serially executes each path. When the warp is executing a given path, all threads that have not taken that path are disabled. If the control flows converge again, the warp is able to return to a single, parallel execution of all threads. Each multiprocessor executes warps much like the *Single Instruction Multiple Data* (SIMD) paradigm, as every thread is assigned to a different SP and every active thread executes the same instruction on different data. The MT-Issue unit weaves threads into warps and schedules an active warp for execution, using a round-robin policy with aging.

Streaming multiprocessors are in turn grouped in Texture Processor Clusters (TPC). Each TPC includes three streaming multiprocessors in the GT200 architecture.

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 2. This shared memory is organized in 16 banks that can be concurrently accessed, each bank having a single read/write port.

B. CUDA Programming Model

The Compute Unified Device Architecture (CUDA) [13], [14], 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 executed by a user-specified number of threads. Every CUDA kernel is explicitly invoked by host code and executed by the device, while the host-side code continues the 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 message-passing, 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.

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 for each thread to handle only small chunks of data.

IV. Design of XTS Encryption Mode for GT200 Architecture

In this section we address the task of engineering the encryption primitive implemented in TrueCrypt to exploit the computational power of the GT200 architecture. We firstly analyse the algorithm in order to highlight the potentially parallelizable sections and identify the parallelization strategies, and then we explore the parameter space proper of both the target application and the chosen hardware architecture, to select the most efficient solution. The main parameters exposed by the CUDA programming model are the allocation of data to the memory hierarchy of the device and the packing of threads into thread blocks.

During the following discussion, we will focus on a single choice for the encryption function E, i.e. Twofish, but similar considerations may be done for both the AES and Serpent algorithms.

Algorithm IV.1: Parallel TrueCrypt XTS Encryption Mode Algorithm.

Input: $P = (P_d, P_{d+1}, \ldots, P_{d+t})$: input plaintext chunk as a sequence of t data units. [d, d+t]: range of data unit indices of the whole plaintext to be encrypted through multiple calls to the algorithm. k_1: primary encryption key. k_2: secondary key.
Output: $C = (C_d, C_{d+1}, \ldots, C_{d+t})$ output ciphertext chunk.
Data: $E(p, k)$: symmetric encryption function on plaintext p using key k. $t = 512$: number of data units in a plaintext chunk.
$m = 32$: number of cipher blocks per data unit. $P_n = (P_{n,0}, \ldots, P_{n,m-1}), C_n = (C_{n,0}, \ldots, C_{n,m-1})\ \forall\ n \in [d, d+t]$.
$N = (N_0, N_1, \ldots, N_t)$: encrypted data units indices.

```
1  begin
2      memCopyToGPU(P,d,k1,k2)
3      foreach j ∈ [0,t] do
4          N_j ← E(d + j, k2)
5      foreach j ∈ [0,t] do
6          foreach i ∈ [0,m − 1] do
7              w_{d+j,i} ← N_j ⊗ α^i
8              C_{d+j,i} ← E(w_{d+j,i} ⊕ P_{d+j,i}, k1) ⊕ w_{d+j,i}
9      memCopyToHost(C)
10     return C
11 end
```

A. Parallelization Strategy

To maintain compatibility with the TrueCrypt infrastructure, and to avoid the loss of performance and portability due to variable plaintext chunk sizes (see Section II), the described implementation processes plaintext chunks of 256 KB (with the associated unique data unit indices).

Given the algoritm reported in Section II, we identify the innermost loop as fully parallelizable, due to the lack of direct inter-block dependencies. The outermost loop includes, in addition to the innermost loop, also the computation of $\bar{n}$. Since there is no loop-carried dependency, we can apply the *loop fission* transformation [5] to split the outermost loop in two loops, the first computing all values of $\bar{n}$, and the second computing the innermost loop. Algorithm IV.1 shows the corresponding parallelized code. The advantage of Algorithm IV.1 with respect to the original Algorithm II.1 lies in the fact that the entire outermost loop is now parallelizable (as highlighted by the use of `foreach` constructs), allowing the implementation to be composed of just two CUDA kernels, one for the computation of the N_j encrypted data unit indices, and one for the encryption of the cipher blocks.

Algorithm IV.2: Parallel TrueCrypt XTS Encryption Mode Algorithm.

1 **begin**
2 memCopyToGPU(P,d,k_1,k_2)
3 **foreach** $j \in [0, t]$ **do**
4 **foreach** $i \in [0, m-1]$ **do**
5 $w_{d+j,i} \leftarrow E(d+j,\ k_2) \otimes \alpha^i$
6 $C_{d+j,i} \leftarrow E(w_{d+j,i} \oplus P_{d+j,i},\ k_1) \oplus w_{d+j,i}$
7 memCopyToHost(C)
8 **return** C
9 **end**

However, this construction still use two different CUDA kernels. Since setting up the execution of a kernel leads to a large overhead, it is worth investigating whether it is possible to obtain an implementation using a single kernel. Consider the original Algorithm II.1. If we replace the computation of a single $\bar{n}$ (line 3) with the computation of m identical values $\bar{N} = (\bar{n}, \bar{n}, \ldots, \bar{n})$, we obtain two innermost loops with the same loop bounds (from 0 to $m-1$). We can now apply the *loop fusion* transformation [5] to these innermost loops, to obtain Algorithm IV.2. While such an algorithm performs the computation of each $\bar{n}$ m times, which may seem to lead to performance degradation, this degradation is limited both by the parallel execution of these computations and by the fact that the computation of these values eliminates the need to load them from memory, which would have otherwise led to a large number of concurrent and conflicting memory accesses. On the other end, having a single CUDA kernel greatly reduces the invocation overhead.

Regardless of the parallelization strategy chosen, the invocation of the encryption primitive requires a data transfer from the host main memory to the device global memory, including the plaintext chunk, the index of the first data unit, and the primary and secondary encryption keys. Conversely, on completion of the encryption, the computed ciphertext is transferred back to the host main memory.

B. Memory Allocation Design

As reported in Section III, when engineering an algorithm for the GT200 architecture, a critical design decision is the allocation of data onto the different memories provided by the architecture, which in turn affects the way threads are split onto thread blocks. We also recall that every thread allocated to the same thread block may access a fast, 16 KB shared memory, while threads in different blocks may only communicate through the slower (but much larger) global memory, endowed with a read only cache. Multiple thread blocks may be allocated by the CUDA scheduler to the same multiprocessor, thus implicitly sharing the same hardware resources, including the shared memory.

In addition to the data described in Algorithm IV.1 and IV.2, we need to consider the data involved in the computation of the encryption algorithm, including both constants and temporary values. Like most symmetric key encryption algorithms, Twofish employs substitution boxes (*S-boxes*), that are frequently accessed during the encryption process. It is important to note that these accesses are non-local by design: temporally consecutive accesses do not refer to spatially adjacent memory locations. Specifically, Twofish employs 4 key dependent S-boxes, each 8 by 8 bit wide. To achieve a performance speedup, these S-boxes are expanded through precomputing the key dependencies and are thus used in an expanded form which occupies 4 KB of memory. In addition to the S-Boxes, Twofish uses a key expansion mechanism which outputs 160 bytes of key material; these data should also be stored in a low latency memory. Therefore, the total amount of data that should be loaded in a shared low latency memory is $160+4096$ bytes, which allows us to use the shared memory, thus exploiting the low latency access provided by this explicitly accessed cache.

Note that constant values, such as S-boxes, could also be stored in the constant memory. Unfortunately, the access to the constant memory cache is single ported, in contrast with the 16 ports exposed by the shared memory. It is therefore sensible to choose the shared memory to store these constants since they will be accessed simultaneously by all threads. To save memory, the key for the data encryption is stored in the same space occupied by the index encryption key once the index encryption has ended. This allows the implementation to occupy less than 1/3 of the shared memory for each thread block, thus allowing multiple CUDA thread blocks to keep their data on the shared memory at the same time. This, in turn, enables faster context switching among the thread blocks on the same multiprocessor.

C. Thread Packing Strategy

In principle, the number of threads per CUDA thread block is constrained by the width of the SIMT stream multiprocessor, which process 32 threads at the same time executing the same instruction. Thus, to maximize the usage of resources, the number of threads per block should be a multiple of 32. A further constraint imposed by the CUDA framework sets an upper limit of 512 threads per block. Since the threads in a

block are divided in *warps* of 32 threads at the architectural level, and all threads in a warp concurrently execute the same instruction, to allow a degree of interleaving of different warps to hide memory latencies, it is advisable to have at least two warps per block, thus imposing a lower bound of 64 threads per block.

Having determined the four options of 64, 128, 256 and 512 threads per CUDA block, we need to explore this design space to determine the best choice for both parallelization strategies.

Once the number of threads per block t has been determined, we can establish the number of CUDA thread blocks as $b = \frac{e}{t}$ where e is the number of cipher blocks in a plaintext chunk (fixed at 2^{14} by the TrueCrypt framework).

V. Experimental Results

In this section, we conduct a thorough experimental campaign to validate the analysis presented in Section IV and provide a performance evaluation of the optimal solution identified.

A. Experimental Setup

The experimental setup consists of a host system based on an Intel Core 2 Quad Q6600 clocked at 2.4 GHz, with an 8 MB L2 cache, and an NVidia GTX260 with 192 processing cores (equivalent to 24 stream multiprocessors) and 896 Mbytes of on-board GDDR3 memory. The CUDA toolkit used is version 2.1, installed on a Gentoo Linux system compiled for x86_64 architecture. The host system offers a PCI-Express version 1 to connect the board.

All the tests have been conducted through issuing write requests on volumes with sizes ranging from 5 MB to 1 GB, to simulate various possible disk buffer sizes provided by modern Operating Systems to cluster physical write operations. The results are gathered as the mean of 30 trials, with a mean square error lower than 1% and were collected using realtime clock sampling primitives in order to achieve the maximum precision available from the system. For the GTX260 board, the timings were taken both with and without the memory transfer operation to and from the device, to evaluate the overhead.

B. Thread Packing Design Space Exploration

The first batch of experiments explores the range of possible numbers of threads per CUDA block, from 64 to 512. The results shown in Figure 3 show that 256 threads represent the optimal choice for all plaintext sizes. This result is justified if we consider that, by architectural constraint, each stream multiprocessor has an issue queue that can accomodate 24 warps of 32 threads each. A block of 256 threads is composed by exactly 8 warps, which is the maximum power of 2 that is also a divider of 24 – a power of 2 is needed to allow each block to have a number of threads that divides the number of cipher blocks in the plaintext chunk, while having a divider of the number of warps allocated in the issue queue allows the issue unit to switch among different blocks with minimal overhead.

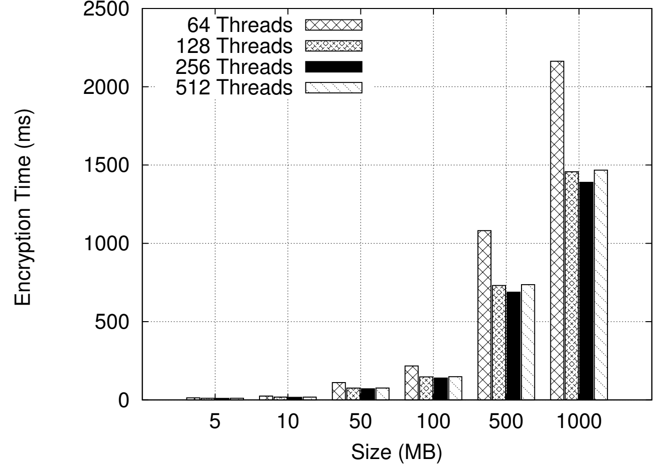

Fig. 3. Comparison between different number of threads per block

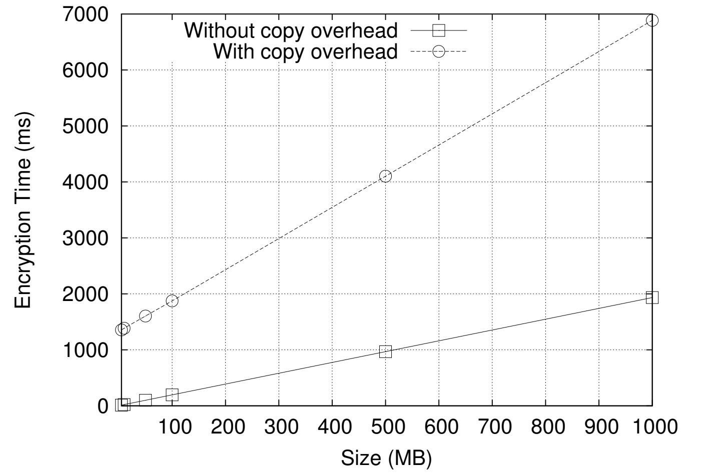

Fig. 4. Comparison between timings with and without the memory transfer overheads

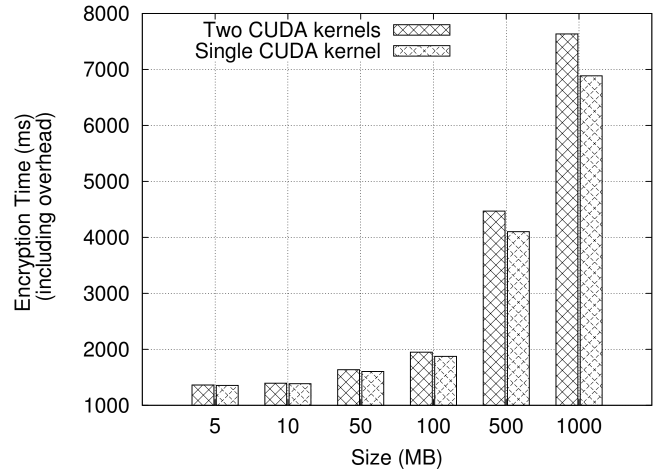

Fig. 5. Comparison between two kernels and a single one

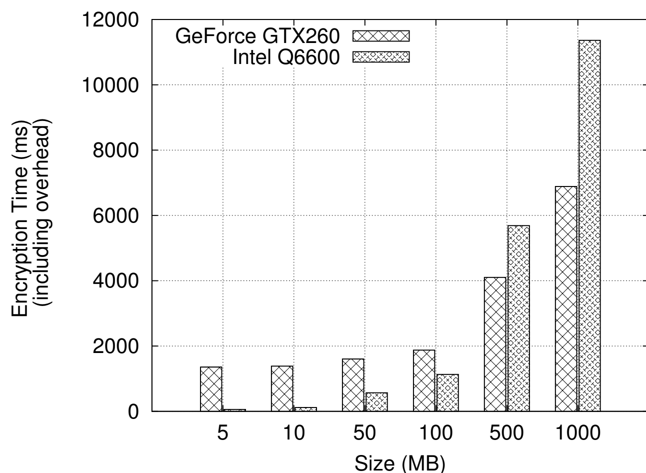

Fig. 6. Comparison between GPU and a quad core CPU

While these considerations might have led to identifying 256 as the optimal choice without exploration, one must take into account the fact that there are many temporary variables which the CUDA compiler may or may not spill to the shared memory, as well as the fact that a small part of the shared memory is used for system purposes by the CUDA framework, making the memory size not a power of 2.

C. Setup and Data Transfer Overhead Analysis

After confirming the optimal number of threads per block to be used, we observed that the setup time and the memory transfers represent a significant cost when using the accelerator. As shown in Figure 4, there is both a fixed overhead which is due to the setup time necessary to wake up the board and a memory transfer overhead which grows linearly with the number of plaintext chunks processed. The fixed overhead amounts to 1326.26 ms while the linearly growing one is quantified in 0.86 ms per encryption primitive call (i.e. for transferring forth and back 256 KB of plaintext). Note that the memory tranfers cannot be entirely removed, but their cost might be reduced using PCI-Express 2.0, which supports twice the bandwith of version 1.

D. Comparison of Parallelization Strategies

Figure 5 shows the comparison of the two parallelization strategies identified in Section IV-A. We can see that the winning strategy is having a single kernel as shown in Algorithm IV.2. This means the increased contention for the shared memory and the kernel invocation overhead due to the split kernel of Algorithm IV.1 degrade the performance more than the increased computational effort undertaken in Algorithm IV.2.

E. Performance Evaluation

Figure 6 compares the performances achieved by our solution with the x86_64 implementation of TrueCrypt. The standard implementation of TrueCrypt is able to exploit parallelism on a multi-core system, assigning plaintext chunks in a round robin fashion to the available cores. The reported test have been conducted using all four cores offered by the host system.

For small sizes of data, the CPUs outperform the GPU, due to the heavy wake up overhead of the device. On heavier workloads, the CPU implementation does not keep up with the performances of the GTX260. The tradeoff point for the system used in the experiments is reached at a plaintext size of 184 MB against four cores, that is 46 MB of encryption per single core. The best speedup is achieved when dealing with one gigabyte wide write requests and it amounts to 67%. We want to underline that TrueCrypt keeps all the cores of the host machine under full load when encrypting the data, while the GPU implementation keeps the system load well under 10% on a single core – mainly due to disk interrupt handling.

The results show a graphic board can be successully employed as a cryptographic accelerator on desktop systems. Tweaking the system buffer size and writeback frequency can help in doing writeback calls nearer to the tradeoff point. In a Network Area Storage system, the GPU implementation can provide a maximum throughput of 152 MB/s against the 92 MB/s provided by the CPU, without lowering the capability of the system to deal with network transfers.

VI. Conclusions and Future Works

In this paper, we addressed the optimized mapping of the Truecrypt disk encryption primitive for the NVidia GT200 architecture, describing how to correctly tune the design parameters, including data allocation, thread packing, and parallelization strategy. Overall, our implementation of TrueCrypt running on a NVidia GTX260 GPU outperforms by 67% the baseline implementation running on a four core CPU. Thus proving the viability of using graphic coprocessors as effective accelerators for disk encryption both for single users and entry- to mid-level network area storage systems. This solution is particularly cost-effective expecially in the enterprise setting where it offers resonable performances without resorting to expensive and not so flexible ad-hoc hardware accelerators.

References

[1] TrueCrypt: Free Open Source On-The-Fly Encryption. http://www.truecrypt.org.

[2] IEEE Standard for Authenticated Encryption with Length Expansion for Storage Devices. *IEEE Std 1619.1-2007*, pages c1–45, 16 2008.

[3] IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, pages c1–32, 18 2008.

[4] Ross J. Anderson, Eli Biham, and Lars R. Knudsen. The case for serpent. In *AES Candidate Conference*, pages 349–354, 2000.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[6] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 334–350. Springer, 2005.

[7] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer, 2002.

[8] Mohamed Abo El-Fotouh and Klaus Diepold. Statistical testing for disk encryption modes of operations. Cryptology ePrint Archive, Report 2007/362, 2007. http://eprint.iacr.org/.

[9] Owen Harrison and John Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2007.

[10] Jim Hughes. IEEE Standard for Encrypted Storage. *Computer*, 37(11):110–112, 2004.

[11] IBM. IBM eServer Cryptographic Hardware Products. http://www-03.ibm.com/security/cryptocards/.

[12] Khronos Group. OpenCL. http://www.khronos.org/opencl.

[13] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, March 2008.

[14] NVIDIA Corporation. CUDA Technology. http://www.nvidia.com/CUDA, September 2008.

[15] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[16] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[17] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[18] Bruce Schneier and Doug Whiting. A performance comparison of the five aes finalists. In *AES Candidate Conference*, pages 123–135, 2000.

[19] Andrew S Tanenbaum. *Modern operating systems*. Prentice-Hall, Upper Saddle River, NJ, 1992.