OpenCL performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives

Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico and Gerardo Pelosi*,†

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Milan, Italy

SUMMARY

The modern trend toward heterogeneous many-core architectures has led to high architectural diversity in both high performance and high-end embedded systems. To effectively exploit the computational resources of such a wide range of architectures, programming languages and APIs such as OpenCL have become increasingly popular. Although OpenCL provides functional code portability and the ability to fine tune the application to the target hardware, providing performance portability is still an open problem. Thus, many research works have investigated the optimization of specific combinations of application and target platform. In this paper, we aim at leveraging the experience obtained in the implementation of algorithms from the cryptography domain to provide a set of guidelines for modern many-core heterogeneous architecture performance portability and to establish a base on which domain-specific languages and compiler transformations could be built in the near future. We study algorithmic choices and the effect of compiler transformations on three representative applications in the chosen domain on a set of seven target platforms. To estimate how well the application fits the architecture, we define a metric of computational intensity both for the architecture and the application implementation. Besides being useful to compare either different implementation or algorithmic choices and their fitness to a specific architecture, it can also be useful to the compiler to guide the code optimization process. Copyright © 2014 John Wiley & Sons, Ltd.

Received 21 March 2014; Revised 17 July 2014; Accepted 19 July 2014

KEY WORDS: heterogeneous architectures; OpenCL; performance portability; implementation space exploration

1. INTRODUCTION

The recent trends in computing systems are leading toward a convergence of high performance and high-end embedded systems; while at the same time, Moore's law is slowing down significantly. This scenario has led to the rise of heterogeneous platforms on the basis of a pairing of multicore general purpose host processors and many-core, massively parallel computing devices [1]. While these platforms are able to provide massive computational power, programming applications to efficiently run on them is significantly more difficult than programming general purpose single or multicore platforms. The Open Computing Language (OpenCL) [2] is a standard for the development of parallel applications on a variety of heterogeneous multicore architectures. A programming model for such architectures needs to exploit the available hardware resources through language constructs that encode the hot-spot computations of the application. OpenCL code will run on a number of widely different heterogeneous computing platforms. However, this does not guarantee that the same code will run with good performance on the same range of platforms – actually, this

*Correspondence to: Gerardo Pelosi, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133, Milan, Italy.

†E-mail: gerardo.pelosi@polimi.it

almost never happens. Thus, the topic of performance portability has become a key challenge to reduce the growing complexity of parallel application programming.

One of the prime platforms used as a target for the OpenCL language is the general-purpose computation on graphics processor units (GPGPUs), which provide a number of execution units in the thousands range, coupled with a high bandwidth dedicated memory [3]. This, in turn, fostered the mapping of a broad range of computationally demanding applications on GPUs and pushed toward the rapid improvement of their programming flexibility [4]. The Compute Unified Device Architecture (CUDA) [5] and the ATI Stream [6] programming models provide explicit language features to specifically exploit the computing power of the nVidia and AMD GPUs, respectively. CUDA and ATI Stream are closely tuned for a specific hardware architecture. Therefore, at the expense of portability, they allow to develop code with better performance than the functionally equivalent OpenCL one running on nVidia and AMD platforms, respectively.

OpenCL is a set of language (OpenCL-`C`) and API (OpenCL-API) that allows programs to be organized into a host part and a device part. The host part usually run on a general purpose processor or multi-processor, and it is in charge of running control-intensive code (*kernels*), as well as offloading compute-intensive code to the device(s). Typically, these kernels are perfect loop nests, where each iteration of the innermost loop constitutes a work-item, and work-items are grouped into work-groups according to an orthotropic geometry. Work-groups are fully independent, while work-items in a work-group can be synchronized with *barrier* constructs.

In developing OpenCL applications, the programmer is in charge of choosing the target device among the ones available on the heterogeneous platform and is given full control of it, including the explicit management of the memory hierarchy, as even small variations in the architectural parameters may lead to a significant performance loss. In addition, a lot of necessary boilerplate code (e.g., for setting up a device or transferring data to it) hinders the readability and maintainability of the core algorithm. In order to mitigate the aforementioned issues, some OpenMP-like extensions to `C`, `C++`, and `Fortran` have been proposed (e.g., OpenHMPP [7] and OpenACC [8]) to support the programmer by new and specialized syntax constructs. However, they cannot yet compete with the performance of equivalent OpenCL code.

Open Computing Language allows code portability (among platforms supporting the same revision of the specifications), which in particular allowed us to run the same OpenCL code benchmarks on all the tested platforms. However, efficient code optimization methodologies effective enough for being automatically applied on different computing platforms are still an open problem.

Tackling the problem at the compiler level has proven difficult, because of the ubiquitous use of target-specific information in OpenCL code. An alternative approach is the use of domain-specific languages (DSL) to capture the features of the target platforms and encapsulate them in high level constructs that easily relate to the domain knowledge of the application programmers. Both these approaches are currently the object of research but neither has already obtained sufficient results to make it the prime candidate for solving the problem of performance portability.

In the last 5 years a large research effort was spent on manually optimizing individual applications on one or more GPU platforms. In the field of cryptography, for example, such an effort has been conducted on the Advanced Encryption Standard (AES) algorithm [9–11], leading to significant speedups over traditional CPU implementations. However, analyzing a single algorithm does not produce results that are of significant help to either DSL designers or compiler writers, as these experiences do not have the breadth to cover an entire application domain.

In this paper, we aim at leveraging the experience obtained in the implementation of algorithms from the cryptography domain on GPGPU architectures to provide a comprehensive set of guidelines for modern GPGPU performance portability and to establish a base on which DSL and compiler transformations could be built in the near future. To this end, we evaluate the effects of a set of code transformations on multiple OpenCL supporting platforms, performing the lowering of the same code with the OpenCL compilers provided by the suppliers, which do not allow the direct implementation of such transformations within them.

1.1. *Contributions*

We demonstrate a more fundamental impact of the parallel programming model with respect to what was exposed in previous literature. While we confirm that interactions between language and architecture are critical, we add a new aspect, the need to tune the design of the algorithm itself to the other two features. Indeed, we show algorithmic choices to be critical, and closely related to architectural features.

To this end, we investigate a specific application domain, in part because algorithm design considerations require application domain knowledge and in part because a single domain can be explored within the constraints of a single article. In particular, we adopt the cryptography domain as our case study. Applications in this domain have already demonstrated good parallelization, through the availability of data parallelism with almost no control divergence. Other interesting features include the presence of countable loops, the possibility to formulate the algorithm using either bitwise or wordwise operations, as well as the use of constant pools of significant size, thus exercising architectural features such as register bank, local memory, and instruction cache.

We choose the Data Encryption Standard (DES), KEELOQ and MD5-crypt ciphers for a mix of interesting properties, including their wide use in real-world applications: DES derivatives are still in use in the electronic payment industry, MD5-crypt is used for password encryption in many UNIX-based operating systems, and KEELOQ has wide application in remote keyless entry systems. Moreover, there are significant applications for implementation of cryptographic primitives targeting GPGPUs. Foremost is brute forcing, that is, the discovery of the secret key by attempting all possible guesses. This may have legitimate applications (e.g., recovering the secret key of KEELOQ having lost the remote controller of the car) as well as less legitimate applications of password breaking – which are still quite important, as demonstrating their feasibility prompts the implementation of more password encryption methods. Another important application of GPGPU implementation of cryptographic algorithms is the encryption/decryption of large quantities of data, such as in disk or volume encryption [12].

We quantitatively compare the performances of OpenCL implementations of each of the three algorithms on seven different target platforms. In addition, we also show the difference in sustained performance among nVidia and AMD GPGPUs. Because OpenCL enables both types of GPGPUs to run the same high level C/C++ code, a fair comparison of their performance can be performed.

As a key result, we provide design best practices for the cryptography application domain, tuned to each relevant architectural feature. We highlight which modifications are needed to the code and the algorithm to achieve performance on each of target platform, paving the way for several future directions, which include the encoding of the best practices identified in this works in a set of templates, which could be used to reduce the redundancy generated by the need to keep multiple versions of the code, as well as serve as patterns for the optimization of new cipher implementations, and the definition of compiler transformations for portability of performance.

1.2. *Organization of the paper*

In Section 2, we describe the architecture of a modern GPGPU; while in Section 3, we introduce the primary optimization techniques employed in this work. Section 4 introduces the cryptographic algorithms employed in the study. Section 5 presents the performance analysis of the cryptographic algorithms on the target GPGPU platforms, introducing a unifying metric to evaluate the affinity of the implementation to the underlying hardware. Section 6 provides an overview of the related literature. Finally, Section 7 draws the conclusions and points out future research directions.

2. MODERN GRAPHICS PROCESSOR UNIT ARCHITECTURES

Many-core architectures offer large amount of parallel computing power by supplying the developer with hundreds of processing cores, each endowed with limited resources. In GPGPU processing elements, the following two resource constraints are most prominent.

Control flow divergence Multiple divergent control flows can be handled safely from the point of view of functionality but with major performance losses as parallelism is inhibited along the different control flows. Essentially, divergent flows of control are serialized, regardless of the data dependencies among the divergent threads (which may well be nonexistent) [13]. This limitation is due to the hardware design of GPGPU, where the processors in a multiprocessor unit are bound to the same program counter.

Local memory availability Limited amount of very fast local memory must be shared among numerous processing elements [14]. While the sharing allows fast communication among the processing elements, the local memory is much more useful when used in a read-only way or partitioned for local use by each processing element, because true shared accesses still require costly synchronization operations and are often difficult to code.

To benefit of the computational features available on a target parallel platforms, the critical issue is to be able to express a given application or algorithm in a form amenable to the parallel execution. The literature reports three main sources of parallelism, which can be exploited with different degrees of success on various types of parallel architectures:

Thread-level parallelism It is obtained when two or more tasks (regions of code with independent control flows) can be carried out simultaneously with either a few or no data dependencies. In the former case, synchronizations will be needed within each task; in the latter case, the synchronization point will be simply put at the end of each task. Thread-level parallelism is exposed by complex applications, where multiple independent tasks are performed, and is best exploited on symmetric multiprocessors, where each processor is endowed with sufficient resources to execute the task assigned to it. This kind of parallelism is not suited for GPGPUs, as control-flow divergence is a major factor for performance reduction [13].

Loop-level parallelism It is found in parallel loop constructs, where each iteration of the loop is either data independent from the others or has limited synchronization requirements. Loop-level parallelism is an excellent fit for vector processors, Single Instruction Multiple Data (SIMD) processors and GPGPUs, because the control flow is fixed and identical for all iterations (barring nested conditionals, which can often be transformed to predicated code) [14].

Instruction-level parallelism It is achieved at the finest of the three common granularities, where independent instructions can be parallelized. It is commonly exploited by superscalar and very long instruction word (VLIW) architectures, but, like Thread-level parallelism, it is mostly unsuitable for GPGPUs because of the need to execute different instructions simultaneously, rather than the same instruction on different data. However, some GPGPU architectures, such as the AMD R700 and R800 generations, do use VLIW cores [15, 16].

2.1. OpenCL programming model

Open Computing Language supports primarily data parallelism and to a lesser extent task parallelism. The main programming structure supporting data parallelism is an explicitly parallel function invocation (*kernel*), which is executed by a user-specified number of work items, organized in an *N-dimensional range*. Every OpenCL 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. Thus, task parallelism is also supported by allowing the user to enqueue multiple single work-item kernels for execution, which may be run in parallel. The programmer is provided with a specific synchronizing function call to wait for the completion of the active asynchronous kernel computation.

The OpenCL programming model abstracts the actual parallelism implemented by the hardware architecture, providing the concepts of *work group* and *work item* to express concurrency in algorithms. A work group captures the notion of a group of concurrent work items. Work groups are required to be computable independently so that it is possible to run them in any order (in parallel

Table I. Open Computing Language memory regions [2].

	Global	Constant	Local	Private
Host allocation	Dynamic	Dynamic	Dynamic	None
Device allocation	None	Static	Static	Static
Host access	R/W	R/W	None	None
Device access	R/W	RO	R/W	R/W

or in sequence). Therefore, the synchronization primitives semantically act only among work items belonging to the same work group. Intrawork-group communications among work items use the *local memory* associated with that work group.

Work items belonging to different workgroups must communicate through *global memory*, which is generally mapped to an off-chip memory. The concurrent accesses to local memory by work items executing within the same work group are supported through an explicit barrier synchronization primitive. In addition to the local memory and the global memory, in the OpenCL programming model, each work item may access a *constant memory*, shared by all work items in the kernel, and a *private memory*, which is private to each work item. Table I summarizes the allocation and access capabilities of both host and devices on the four OpenCL memory address spaces. A particular aspect to be noted is that the memory allocation on the device should be statically performed, as it is rather common for the computing devices to be lacking a proper memory management unit.

A kernel call site must specify the number of work groups and the number of work items within each work group when executing the kernel code. Contrary to proprietary models such as CUDA [17, 18], OpenCL does not impose at the language level predefined limits on the size of the *N*-dimensional ranges. It relies on a platform introspection API to allow the programmer to retrieve at runtime such limits for each available compute device on the platform, thus allowing more flexibility in the definition of kernels and the ability to support compute devices from multiple vendors and multiple devices attached to the same host.

2.2. nVidia graphics processor unit architectures

nVidia has been the first manufacturer to offer a range of GPU architectures together with a proper runtime support enabling general purpose computing with both OpenCL and their proprietary CUDA programming model. nVidia GPGPU platforms are divided into graphics processors with compute capabilities and full high-performance computing platforms. High-performance computing platforms are marketed under the Tesla brand and are basically a scaled-up version of the common graphics processors, featuring more execution units and a larger dedicated memory. We will consider in our exploration GPUs coming from three different generations of nVidia platforms: the GT200 [19], Fermi [20], and Kepler [21] architectures.

All the nVidia GPU architectures organize a large set of *stream processors* (SPs; simple single-issue computing units with an integer arithmetic logic unit (ALU) and a floating point unit) in groups, called *stream multiprocessors*. In addition to the SPs, the nVidia GPUs also include dedicated units (Special Function Units, SFUs) to compute complex floating point primitives typical of the graphic rendering workload, together with the support for 128-bit wide PTX registers. The structure of the hierarchy, the size of the register file and cache memories, and the instruction issue strategies are the features that evolved across different nVidia architecture generations. All the nVidia GPUs are built as a tiling of stream multiprocessors, together with a memory controller employed to access the GPU card global memory. Figure 1 presents an overview of the structure of the stream multiprocessors of the three different architectures we will consider in this work: the GT200, Fermi, and Kepler architectures. The stream multiprocessors are composed by a group of SPs sharing one or more instruction issue units, a common register file, and an addressable local cache. The local addressable cache is accessible with a 10–20 cycles latency and are employed by the nVidia OpenCL compiler to map the local memory of the work groups. In the GT200 GPU series, there are eight SPs per stream multiprocessor, running at twice the clock of the GPU, while the single issue unit runs at half of the clock frequency of the GPU. The issue unit is shared by all

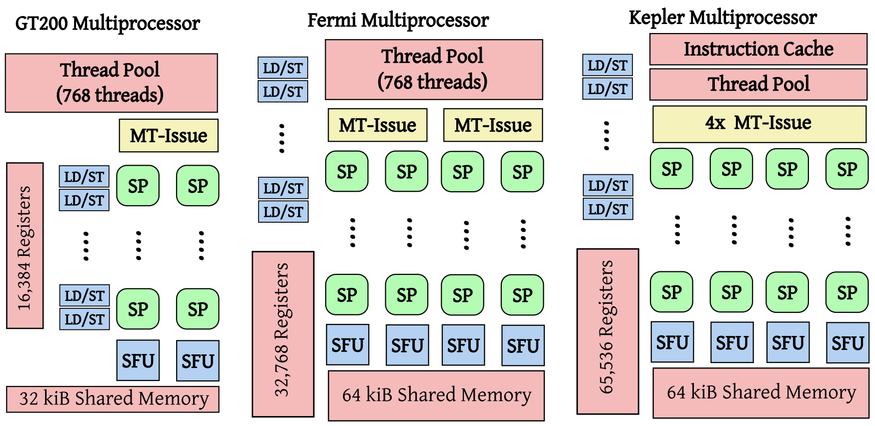


Figure 1. Structure of the stream multiprocessors in the GT200, Fermi, and Kepler architectures.

the SPs; thus, an instruction issued is effectively executed 32 times on different data, yielding what nVidia defined as the single instruction, multiple threads execution model. Consequentially, a set of 32 threads executed on the GPU are commonly called a *warp* in nVidia parlance, and it is strongly advised that work groups running on nVidia devices should include a number of work items multiple of 32 to achieve optimal performances [22]. In case control flow divergences are present in the program, the GT200 architecture simply computes both the branch-taken and the branch-not-taken alternatives, effectively retiring only the instructions depending from the correct condition. We note that the alternatives are executed serially; thus, the presence of control flow divergences affects strongly the performances of the GT200 architecture. The instruction issue unit is able to pick the warp to be issued on the SPs among 24 different ones belonging to at most three different work groups, to hide the possible memory access latency that a warp is undergoing. The memory access operations are performed by 16 load-store units present in the stream multiprocessor of the GT200 architecture.

The Fermi architecture improves its predecessors on multiple fronts: First of all, the number of SPs per stream multiprocessor is raised to 32, and two issue units, running at the full GPU clock speed are present. This in turn implies that 64 instructions coming from two different warps are executed at each clock cycle by a stream multiprocessor. To cope with the larger execution contexts and the higher memory pressure, both the register file and the local shared addressable cache are doubled in size with respect to the GT200 architecture. In addition to this, the Fermi architecture is characterized by a second-level nonaddressable data cache of 768 kiB, which is shared by all the stream multiprocessors, speeding up eventual repeated accesses to the global GPU memory. The introduction of this L2 cache impacts significantly on the performances of the algorithms requiring a working set larger than the registers and local addressable cache, as an access to the global GPU memory takes around 100–200 clock cycles, according to nVidia's specifications [22].

The Kepler architecture pushes forward the improvements introduced by the Fermi architecture, effectively doubling again the number of register available to one stream multiprocessor, effectively reaching 65 536, 32-bit wide registers. The number of SPs per stream multiprocessor is raised to 192; however, these units no longer run at twice the GPU clock frequency, thus resulting in an effective 192 instructions being executed per clock cycle. The number of load-store units per stream multiprocessor is raised to 32, and the number issue units is doubled again, yielding a four-issue unit stream multiprocessor. Each of these four issue unit is actually able to schedule two warps on the associated computation units, picking them among a 64 warp wide instruction buffer. The stream multiprocessor is also endowed with an instruction cache, which was not present in the previous Fermi architecture. At chip level, the improvements of Kepler over Fermi are represented by a larger L2 cache (1536 kiB wide) split equally on the six separate memory controllers. This change in design with respect to the GT200 architecture allows the GPU to cope better with irregular memory access patterns, as the fragmentation of the data has a lesser impact given the six independent memory units. We note that the latest architecture, Kepler, adds support for the *dynamic parallelism* in computation, that is, the ability to spawn new computations from threads running on the device. Such feature, however, is not currently available on OpenCL and thus cannot be investigated yet in our exploration.

2.3. AMD graphics processor unit architectures

AMD GPUs are the principal alternative in the field of off-the-shelf many-core architectures that support the OpenCL programming model. AMD started offers the support for OpenCL as the main programming language to tap into the computing capabilities of its GPUs, although a low level alternative toolkit, Close To Metal was proposed at first. We will analyze the architecture of the R700 [16] and R800 [15] GPU generations by AMD, pointing out the differences from the architecture proposed by nVidia.

The key idea of AMD architectures is exploiting a rather complex SP to provide a fine-grained parallelism handling, reducing the penalties induced by control flow divergences. The SP in AMD architectures is richer than its nVidia counterpart: In fact, a single AMD SP is a VLIW unit endowed with a branch handling lane as depicted in Figure 2. In particular, the VLIW of the R700 architecture is composed by four ALUs and one unit able to compute also transcendental functions, together with a register file of size 16 384. In the R800 architecture, the structure of the VLIW SP was improved through replacing the four ALUs and one general purpose unit layout with a four general purpose unit one. While this reduces the effective parallelism, it increases the flexibility of instruction scheduling on the single VLIW unit, as there are no longer constraints on which lane should execute transcendental functions computations. The change in the structure of the VLIW processor was coupled with an increase in the size of the register file as the 16 384 32-bit registers of the R700 design are substituted with 128-bit ones, effectively quadrupling the register file size. Both the R700 and R800 VLIW units are endowed with bit-set/reset and extraction dedicated instructions, which were introduced to speedup the computation of typical GPU computational loads, such as video codecs.

The SPs are grouped together into *SIMD cores* (the analogue of stream multiprocessors in nVidia's parlance): Each SIMD core contains two groups of 16 SPs. Each of the two groups is endowed with an instruction issue unit and a local shared memory, 8 kiB wide in R700 and 32 kiB in R800. Each SIMD core computes a set of work items coming from a work group and is able to handle up to 960 of them in the R700. By contrast, the number of work items per work group tackled by the R800 stream cores was reduced to 256 for power saving concerns.

It is possible for a SIMD core to exchange information quickly with the others through a globally shared, addressable, 64 kiB wide data cache and a common synchronized register file dedicated to 32-bit atomic access variables. This structure was designed to allow efficient barrier synchronizations in AMD architectures, and there is no explicit counterpart in nVidia's platforms. The

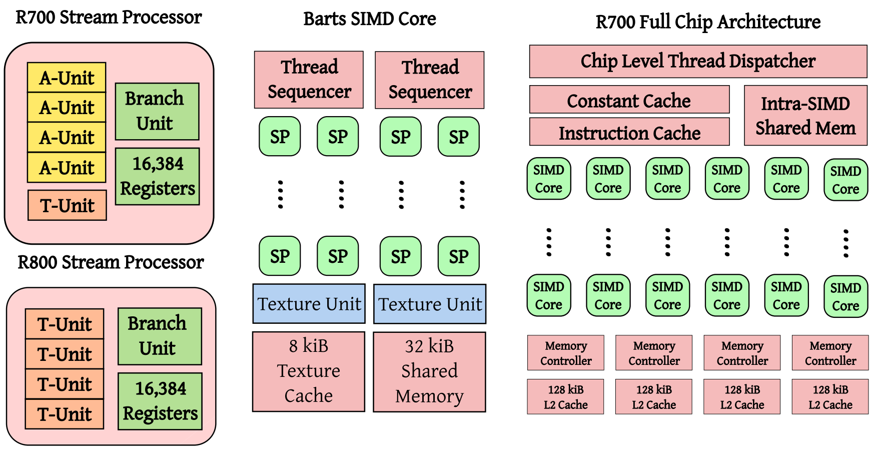


Figure 2. Structure of the stream processors and stream multiprocessors in the R700 and R800 architectures.

communication with the device global memory, that is, the video RAM, is managed by four memory controllers, each one endowed with 128 kiB wide cache bank and a 64-bit data bus each. This memory controller structure allows to cope with different read operations acting on data scattered in the global memory with greater ease with respect to a single 512-bit wide controller.

3. OPTIMIZATION TECHNIQUES

In this section, we present five programming and compilation techniques that have proven useful in the optimization of cryptographic primitives. For each technique, we discuss its applicability to the two families of GPGPUs presented in Section 2.

3.1. Loop unrolling

Loop unrolling is a classic compiler transformation, employed primarily not only to enlarge basic blocks and provide more opportunities for instruction level parallelism but also to reduce loop management overhead and improve data locality [23]. It consists in a restructuring of the loop iterations so that the instructions corresponding to multiple iterations of the original loop are performed in a single iteration of the modified one, which thus performs a reduced number of iterations, at the cost of an increase in the code size.

In the context of GPGPUs, the ability to reduce branch control instructions is critical, because these operations are more costly than in traditional CPUs. However, in the most recent GPGPUs, instruction caches have been introduced. Thus, there is a trade-off between savings induced by the lower number of branches and reduced effectiveness of the instruction cache because of enlarged code. This trade-off is most notable in AMD GPUs, which are better able to handle control flow, thanks to their architecture, which is more similar to a typical multiprocessor with respect to the SIMT model followed by nVidia.

3.2. Bit slicing

Some algorithms may need to perform a large amount of operations on individual bits of the values involved in the computation. These operations are usually ill-suited to software execution, as only a few platforms have dedicated bit-set and bit-clear instructions. To this end, Eli Biham in [24] was the first to describe a software optimization technique, proposed to obtain an efficient DES implementation, which allows to compute operations acting on individual bits efficiently in software. The key idea of the so-called bit slicing technique is to consider bitwise operations among n-bit integers as a single bit operation, executed by n virtual processors, each one computing the same instruction different data bits. This in turn implies that the common n-bit bitwise Boolean operation units of the ALU are seen as vector units effectively performing n, 1-bit, Boolean operations.

The required transformation to transform the code into a bit sliced version starts from the decision of the degree of bit slicing that should be employed, which typically is the same as the word size of the computing architecture. To fully exploit a bit slicing transformation, the code should exhibit a data parallelism level at least equal to the bit size of the computing architecture word: This is true in our case studies, which have a significantly high data parallelism. Under the aforementioned assumption, the input data is partitioned into blocks, which represent the input of a single algorithm run. Subsequently, the blocks are encoded, n, at a time in a format suitable for bit sliced computation. The encoding stores all the ith bit of the n input blocks in the same register, packing them together, effectively employing the register as a vector register for 1-bit values. Consequentially, an operation acting on the content of two registers is actually being performed n times on the ith bit of all the input values, without the need to employ bitmasks and shift operations to extract single input bits. In particular, the effects of bitwise shifts and rotate operations on the inputs are performed through simple register renaming, thus removing them completely. This technique effectively reduces the number of instructions to be performed at the cost of a sharp increase in the register pressure caused by the computation, namely, a multiplication by the bit width of an input block. A key point to be noted is that applying bit slicing to nonbitwise operations (e.g., arithmetic addition, and multiplication) may result in a significant performance loss, as the bit-level expression of such operations is complex, and computing it in a bit sliced fashion would imply a higher number of operation than just performing the multiple additions.

The literature reports expected speedups around $5\times$ [24], for a bit sliced implementation of the DES algorithm, considering its implementation on general purpose CPUs, which are endowed with a rather limited amount of registers, and thus are typically affected by a significant number of spills. Therefore, performing bit slicing on a GPU, which is characterized by a remarkably large register file, the improvements brought in by the transformation should be more significant.

We report the detailed implementation of the bit sliced KeeLoq cipher in Appendix A

3.3. Exploiting addressable caches

All the current GPGPU architectures offer a small amount of explicitly addressable cache, which is usually accessible with latencies an order of magnitude smaller than the global device memory. Their usage has been explored since the very first works in the field [17, 25], and a proper exploitation of their presence is paramount to achieve a properly performing implementation. Typically, the OpenCL frameworks distributed by the producers employ the explicitly addressable caches to map the local memory of the programming model, exploiting the eventual remainder from the mapping to make up for an eventual the lack of registers required to map the work-item private variables. In this work, we will provide an analysis of the differences in the architecture design choices made by AMD and nVidia in terms of placement of the addressable caches in the shared memory architecture and the consequent effects on the computation. The key difference between the two architectures is that nVidia has preferred to place the explicitly addressable caches very close to the register file, thus obtaining a negligible access latency, while AMD chose to add an extra addressable cache that is farther from the computation units but accessible by all of them. The advantage of AMD's choice is the ability to efficiently support the barrier construct of the OpenCL programming model by means of the extra addressable cache.

3.4. Register pressure trade-offs

Given the amount of parallelism available in a work group, it is to be expected that equally large amounts of storage resources will be needed. In nVidia GPUs, there is little difference in terms of performance between registers and the explicitly addressed shared memory (which is exposed in OpenCL as a local memory). Thus, if more registers are needed, it is possible to allocate some variables in the OpenCL local memory, without incurring in major penalties for spilling into the (slow) global memory. On the other hand, the explicitly addressed shared memory available in AMD GPUs has a longer latency.

Moreover, in recent GPU architectures, L2 caches are also available. While the nVidia architectures feature a standard L2 cache, AMD has a banked L2 cache, which is more efficient when accesses are uniformly distributed over the global memory, much less so when there is locality.

When the two aspects are combined, the result is that nVidia architectures are much better at dealing with spills than AMD ones. Thus, AMD architectures are more sensitive to register pressure.

3.5. Vectorization and vector types

The OpenCL language encompasses native integer and floating point vector types: these data types provide a mean to the programmer to explicit data parallelism in terms of expressing operations among the elements of two vectors in a natural way. It is pretty common for GPGPUs programming handbooks to recommend vector types usage [26] as an optimization technique from which any implementation will benefit. The use of vector types can be seen as complementary to a proper workload split in work items and work groups. In particular, in case the target platform has an efficient thread spawning strategy, a more effective partitioning of the workload will play a key role, reducing the importance of vector types. On the other hand, in cases, such as Intel's OpenCL back end, which exploits the multicore CPUs of the host as an execution unit, we expect the use of vector types to be strongly beneficial. This is to be ascribed to the fact that they are directly mapped into

ALGORITHM 4.1: KEELOQ Encryption [27]

Data: The function $\texttt{NLF}_{\texttt{T}}(\texttt{idx})$ maps the 5-bit value of the input idx to a single bit tabulated in T,
idx=$\langle i_4, i_3, i_2, i_1, i_0\rangle$, with $i_0, \ldots, i_4 \in \{0, 1\}$,
T=$\langle \texttt{T}_{31}, \ldots, \texttt{T}_0\rangle$=0x3A5C742E, is the 32-bit constant employed to fetch the indexed bit

Input: P=$\langle \texttt{P}_{31}, \ldots, \texttt{P}_0\rangle$: 32-bit plaintext string, K=$\langle \texttt{K}_{63}, \ldots, \texttt{K}_0\rangle$: 64-bit cipher-key string

Output: C=$\langle \texttt{C}_{31}, \ldots, \texttt{C}_0\rangle$: 32-bit ciphertext string

```
state ← P // 32-bit plaintext into the right-shift register of the cipher-state
key ← K // 64-bit key into the right-shift register of the cipher-key
for cnt ← 1 to 528 do
    $s$ ←NLF_T(state_31, state_26, state_20, state_9, state_1) // Output bit of the NLF
    tmp ← $\texttt{state}_0 \oplus \texttt{state}_{16} \oplus \texttt{key}_0 \oplus s$
    state ← state>>1
    $\texttt{state}_{31}$ ← tmp

    tmp ← $\texttt{key}_0$
    key ← key>>1
    $\texttt{key}_{63}$ ← tmp
$\langle c_{31}, \ldots, c_0\rangle$ ← state
return $\langle c_{31}, \ldots, c_0\rangle$
```

the multimedia extension vector registers of the CPU (the ones of Streaming SIMD Extensions units in particular), effectively yielding a significant performance increase.

4. CRYPTOGRAPHIC ALGORITHMS

In this section, we will provide the background on the chosen cryptographic algorithms, highlighting their structure, to point out which optimizations will be improving their performances on GPUs. The choice of these three ciphers was motivated by them being designed either for hardware implementation only or to explicitly slow down the execution in software.

4.1. Data encryption standard cipher

The DES (DES) cipher is a symmetric block cipher with 64-bit block size and a 56-bit key [27].

The DES consists of two parts, the encryption/decryption algorithm and the key-scheduling algorithm. The DES encryption/decryption algorithm is an iterated block cipher consisting of 16 rounds, each designed with a Feistel structure, which processes the left half of the block with through a nonlinear function, the so-called Feistel function, and combines the result via xor with the right half of the block. This result is employed as the right half of the input to the next round, while the old value of the right half is used as the left half of the block input to the next round [27]. The nonlinear function, also known as Feistel function, performs an initial expansion on the 32-bit value obtained in input, resulting in a 48-bit value, to which 48 bits of the secret key are added via xor. The result is split into eight portions, each of them 6 bit wide, to which a 6-to-4-bit nonlinear function is applied. The 48-bit round keys are obtained via a key schedule algorithm that at first permutes the key bits (permuted choice 1), discards the eight parity bits, and divides the key material into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (depending on the round), and then, 48 *subkey* bits are selected through a second fixed permutation (permuted choice 2). A different set of key bits are used in each subkey (one for each round of the encryption/description algorithm) in such a way that each bit is used in 14 out of the 16 subkeys.

The cipher starts processing the 64-bit plaintext via an initial bitwise permutation (IP) of its bits, the output of which is divided into two 32-bit blocks in order to serve as input of the first round. Following this 16 rounds are computed as described earlier, and the left and right halves of the results are swapped after the end of the computation. The result is thus subject to a final bitwise permutation ($\texttt{IP}^{-1}$) to generate the 64-bit ciphertext.

From a software execution point of view, the DES block cipher is characterized by a large number of bitwise operations, which have intentionally been designed to be hard to execute on software platforms. On the other hand, the register pressure exerted by the algorithm is particularly low, and the involved constants are small, thus providing minimal slowdowns due to cache misses.

4.2. KEELOQ *cipher*

KEELOQ is the most scrutinized encryption engine used in remote keyless entry systems [28]. It is a proprietary Nonlinear Feedback Shift Register cipher designed as a pair of feedback shift registers (FSR) coupled with a nonlinear function (NLF) that provides its effective security margin. Algorithm 4.1 shows the pseudocode of the KEELOQ encryption primitive: The cipher key is stored in a right-shifted FSR and is at most 64-bit wide; the 32-bit input plaintext is also stored in a right-shifted FSR as initial state of the cipher, while a 32-bit ciphertext is computed for every run of the primitive. At every step of the encryption computation (i.e., at every clock cycle of the encryption circuit), five bits of the state FSR are combined together by means of the NLF to output a single bit, s, per clock cycle. The bit value in s is xor-ed with the least significant bit of the cipher-key register and subsequently also with two bits of the state register (the 1st and the 17th) to compute a temporary bit value: tmp. The tmp value is employed as the feedback bit of the state FSR, while the value of the least significant bit of the cipher-key FSR is employed to feed the most significant one. After 528 updates of both the state and the key FSRs, the content of the state register is output as the final ciphertext.

The KEELOQ cipher key is a device-unique 64-bit string $k_{\mathtt{dev}}$ generated applying the KEELOQ decryption primitive twice to obtain the two 32-bit halves of its value. In most implementations, the KEELOQ cipher key is derived considering the following: (i) an embedded 64-bit master key (which is fixed by the manufacturer of the keyless entry system); (ii) the serial number of the remote-control device ID, *(iii)*; and a random seed composed by 32, 48, or 60 bits.

An attacker may retrieve the master key from the decoding device (receiver) and eavesdrop the ID of the remote-control device from the transmitted packet that includes both the ciphertext and the ID. Therefore, the secret random seed is the only parameter that avoids the leakage of the secret key of the targeted device if the previous conditions hold.

4.3. MD5-CRYPT *hash function*

The MD5-*crypt* algorithm has been the default method for storing login passwords on a large variety of Portable Operating System Interface (POSIX) compliant UNIX-like systems, among which Linux until some years ago, and it is still quite popular today, in particular in legacy systems [29]. The main idea behind storing hashed passwords is that if an attacker gets access to the storage server, he can't easily just get all passwords and start using them. MD5-crypt is built combining a large number of computation of the MD5 hash algorithm employing alternatively as input either a portion of the password or a portion of a random value, known as salt, which is 64-bit wide or more. The main reason for the large number of MD5 computations is the fact that the MD5 hash was designed to be efficiently executable on single core CPUs with a 32-bit wide word. Computing the MD5 hash many times introduces an overhead, which is negligible for the user trying to log on, but it consistently raises the amount of effort an attacker willing to try to brute force the password has to go through. The reason for the addition of the salt, which is stored in clear along with the password hash, is to prevent the storage of precomputed MD5-crypt digests, raising exponentially even a large offline distributed effort. The details of MD5-crypt are reported by algorithm 4.2: It takes as input a random bit string (salt) composed at least by 64 bits and a user passphrase composed also as a bit string with variable length and outputs a 22-char printable string. First of all, the algorithm computes the hash of the concatenation of the password, the salt, and the password again: MD5(pwd||0x243124||salt). Then, it stores in a new buffer b the password followed by the binary string encoding the three characters \$1\$ (0x243124), the salt and as many bytes from the MD5 hash as the byte length of the password. For each bit of the password, if its value is set, a 0x00 character is appended to b; otherwise, the null character is replaced by the first one of

ALGORITHM 4.2: `MD5`-crypt [29]

```
Input: pwd: bit-string of the input pass-phrase, byte-aligned
       salt: bit-string of the input random salt, byte-aligned
Output: $\langle c_0, \ldots, c_{21} \rangle$: MD5-crypt digest, it is a printable char-string
        including symbols among [A−Z], [a−z], [0−9], ., and /
$\langle d_0, d_1, d_2, d_3 \rangle \leftarrow$ MD5(pwd||salt||pwd)
// pwd||salt||pwd: bit-string concatenation. $d_i$:32-bit word, $0 \le i < 4$
$\langle a_0, \ldots, a_{15} \rangle \leftarrow$ SplitWordsInBytes($\langle d_0, d_1, d_2, d_3 \rangle$)
// $\langle \mathtt{pwd}_0, \mathtt{pwd}_1, \ldots, \mathtt{pwd}_{t-1} \rangle$ denotes the input pwd as a bit-string. $t$=BIT-LENGTH(pwd)
// $\langle \mathtt{pwd}^{(0)}, \mathtt{pwd}^{(1)}, \ldots, \mathtt{pwd}^{(t/8-1)} \rangle$ denotes the input pwd as a byte sequence,
//                         with $\mathtt{pwd}^{(j)} = \langle \mathtt{pwd}_j, \mathtt{pwd}_{j+1}, \ldots, \mathtt{pwd}_{j+7} \rangle$, $0 \le j < t/8$
$b \leftarrow$ pwd||0x243124||salt
for $i \leftarrow 0$ to $t/8-1$ do
    $b \leftarrow b || a_{(i \bmod 16)}$ // byte concatenation
for $j \leftarrow 0$ to $t-1$ do
    if $\mathtt{pwd}_j = 1$ then $b \leftarrow b$||0x00 else $b \leftarrow b || \mathtt{pwd}^0$ // byte concatenation
$t \leftarrow$ MD5($b$)
for cnt $\leftarrow 0$ to 999 do
    if cnt mod $2 \neq 0$ then $b \leftarrow$ pwd else $b \leftarrow t$
    if cnt mod $3 \neq 0$ then $b \leftarrow b$||salt
    if cnt mod $7 \neq 0$ then $b \leftarrow b$||pwd
    if cnt mod $2 \neq 0$ then $b \leftarrow b || t$ else $b \leftarrow b$||pwd
    $t \leftarrow$ MD5($b$)
$\langle c_0, \ldots, c_{21} \rangle \leftarrow$ ASCII-Encode($t$) // special base64-type encoding
return $\langle c_0, \ldots, c_{21} \rangle$
```

the password. The initialization phase terminates computing the `MD5` hash of this buffer b and storing it in temporary variable t. The core part of the algorithm is a 1000-round loop that composes a new string buffer through mixing t, the `salt`, and the original password depending on the index of the current iteration. As last operation within the loop, the temporary variable t is updated by the `MD5` hash value of the bit string included in the buffer. Out of the main loop, the hash value in t is encoded into a printable string composed by alphanumeric ASCII symbols plus the ones corresponding to a period character or a slash character. As reported by algorithm 4.2, the `MD5`-crypt primitive computes its intermediate values bytewise, while the `MD5` one processes its intermediate values wordwise, as shown by algorithm 4.3. These design choices makes the whole code harder to optimize, in particular, it's not possible to use vector data types efficiently because the developer has to choose between a memory layout which favors either the byte view or the word view. Indeed, the implementation of an exhaustive search of the password result to be significantly hindered as a bit sliced design of the code is made useless.

In order to give a comprehensive description of the kernel code employed for the `MD5`-crypt password cracking, we now provide a description of the message-digest algorithm `MD5` designed by Ron Rivest in 1992, employed in by the password hashing engine. As shown by algorithm 4.3, the `MD5` primitive takes as input a bit-string `msg` of arbitrary length, zero pads it to obtain a bit length that is equivalent to 448 mod 512 and concatenates to the result the `msg` length encoded as an unsigned 64-bit word; finally, it produces a message digest as a sequence of four 32-bit words acting wordwise on the intermediate values.

5. PERFORMANCE ANALYSIS

In this section, we provide the definition of a unifying figure of merit to evaluate the goodness of fit of an implementation on a specific platform; then, we report the settings of the test benches employed for the implementation-space exploration of the cryptographic primitives, described in

ALGORITHM 4.3: `MD5` [30]

```
Data: $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$: non-linear round functions, each maps three 32-bit words to a single 32-bit word
      $\mathcal{F}_0(a,b,c){=}(a{\wedge}b){\vee}(a{\wedge}c)$, $\mathcal{F}_1(a,b,c){=}(a{\wedge}c){\vee}(b{\wedge}\neg c)$, $\mathcal{F}_2(a,b,c){=}a{\oplus}b{\oplus}c$, $\mathcal{F}_3(a,b,c){=}b{\oplus}(a{\oplus}\neg c)$
      $\mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_2, \mathrm{S}_3$: rotation amount functions, each maps a value $0{\le}i{\le}3$ to a value in $\{0,\dots,31\}$
      $\mathrm{S}_0(i){\equiv}_{32}7{+}5i$, $\mathrm{S}_1(i){\equiv}_{32}i(i{+}7)/2{+}5$, $\mathrm{S}_3(i){=}\textsc{LookUp}_i(\langle v_0,\dots,v_3\rangle)$ with $0{\le}i{\le}3$ and
      $\langle v_0,\dots,v_3\rangle{=}\langle 4,11,16,23\rangle$, $\mathrm{S}_4{\equiv}_{32}(i{+}3)(i{+}4)/2$
      T: round constant function, it maps a value $0{\le}v{\le}63$ to a 32-bit constant value $\mathrm{T}(v){=}\lfloor 2^{32}{\cdot}\sin(v{+}1)\rfloor$
      $\mu_0, \mu_1, \mu_2, \mu_3$: word selection functions, each maps an index $0{\le}i{<}16$ to a word index in $\{0,\dots,15\}$
      $\mu_0(i){\equiv}_{16}i$, $\mu_1(i){\equiv}_{16}5i{+}1$, $\mu_2(i){\equiv}_{16}3i{+}1$, $\mu_3(i){\equiv}_{16}7i{+}1$
Input: msg: input bit-string
Output: $\langle d_0, d_1, d_2, d_3\rangle$: message digest as a sequence of four 32-bit words
$\langle \mathtt{blk}_0, \mathtt{blk}_1, \dots, \mathtt{blk}_{t-1}\rangle \leftarrow$ MD5_PAD(msg)
 // $\mathtt{blk_j}{=}\langle m_{16j},\dots,m_{16j+15}\rangle$, $m_{j+i}$: 32-bit word, with $0{\le}i{\le}15$, $0{\le}j{<}t$
$\langle d_0, d_1, d_2, d_3\rangle \leftarrow \langle$0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476$\rangle$
for $j \leftarrow 0$ to $t{-}1$ do
    $\langle m_0,\dots,m_{15}\rangle \leftarrow \mathtt{blk}_j$ // $\langle m_{16j},\dots,m_{16j+15}\rangle$
    for idx $\leftarrow 0$ to 3 do
        for $i \leftarrow 0$ to 15 do
            tmp $\leftarrow \mathcal{F}_{\mathtt{idx}}\left(d_{(i+1) \bmod 4}, d_{(i+2) \bmod 4}, d_{(i+3) \bmod 4}\right)$
            tmp $\leftarrow$ tmp $+ d_{i \bmod 4} + m_{\mu_{\mathtt{idx}}(i)} + \mathrm{T}(i{+}16{\cdot}\,\mathtt{idx})$
            $d_{(i+1) \bmod 4} \leftarrow d_{(i+1) \bmod 4} + (\mathtt{tmp} \lll \mathrm{S}_{\mathtt{idx}}(i \bmod 4))$
return $\langle d_0, d_1, d_2, d_3\rangle$
```

Section 4, on the chosen platforms. We will provide experimental results on the effects of optimizations described in Section 3, highlighting how they alter the computation/memory transfer balance, together with the consequences on the performances. We will conclude this section providing the best throughput figures for the selected algorithms on the examined architectures.

5.1. Evaluation metric

One of the key issues in understanding the performance of the OpenCL implementation of an algorithm is understanding whether it actually fits the underlying architecture or not. Different architectures even in the same family expose widely different resources, so absolute results are not especially useful in getting this kind of insight. A metric able to (even partially) remove the bias related to the overall amount of resources from the picture would be desiderable. We introduce one such metric, aiming at representing the optimal ratio of computation versus communication in a given heterogeneous parallel platform.

Definition 1 (Hardware computational intensity)
The hardware computational intensity of a parallel accelerator, denoted as $\mathtt{CI}_{\mathtt{HW}}$, is defined as the minimum number of computational instructions that should be executed per second to fully exploit its memory transfer rate. Given the peak computational power of the parallel accelerator and its data memory transfer bandwidth, the $\mathtt{CI}_{\mathtt{HW}}$ index (expressed in 'instructions per byte' (I/B)) is computed as the ratio between the aforementioned quantities and provides a metric of the optimal computational load supported by the device.

Definition 2 (Implementation computational intensity)
The implementation computational intensity of a given kernel code, denoted as $\mathtt{CI}_{\mathtt{kernel}}$, is defined as the ratio between the total number of kernel instructions executed in a run of the application and the size of data transferred to/from the memory. The $\mathtt{CI}_{\mathtt{kernel}}$ index, expressed in I/B, measures the effectiveness of a kernel code on a specific hardware architecture.

If the implementation computational intensity of a kernel instance $\mathtt{CI}_{\mathtt{kernel}}$ is higher than the hardware counterpart of the corresponding platform $\mathtt{CI}_{\mathtt{HW}}$, the kernel performance is computationally bound as the executed instructions will likely be able to mask the latencies of the required memory transfers. By contrast, if the $\mathtt{CI}_{\mathtt{kernel}}$ index is lower than the $\mathtt{CI}_{\mathtt{HW}}$ one, then the kernel execution performance is limited by the memory transfer rate provided by the hardware. This is because the computing units of the parallel accelerator will be idling while the data is transferred to/from the memory.

5.2. *Experimental settings*

We performed our experimental campaign on a platform set encompassing four nVidia GPUs and two AMD ones and employing the CPU OpenCL back end provided by Intel as a further benchmark to enhance the platform diversity.

Table II reports a full summary of the relevant technical specifications of the platforms. nVidia platforms involved in the exploration span across all the three different architectures are described in Section 3, with three GPUs being commonly used in desktops (GeForce GTX 260, GTX 470, and GTX 660 Ti) and a low power, limited bandwidth alternative employed in the notebook range (Quadro 2000M). The two AMD GPUs are representatives of the R700 and R800 architecture lines, as we are interested in exploring the differences in performances induced by the architectural change between the two generations. The host CPU employed to benchmark the OpenCL back end by Intel is a quadcore Core i7-2820QM, clocked at 2.3 GHz, with the possibility of raising the clock up to 3.4 GHz if the thermal budget of the chip allows it. Table II reports, for each platform, the theoretical peak *instruction per second* (IPS) executable by each platform, together with the bandwidth between the device (i.e., either the GPU or the CPU) and the corresponding main memory (i.e., the video RAM for the GPU or the main RAM for the CPU). We note that AMD GPUs are characterized by a significantly higher IPS figure, because the SPs of their architectures are actually able to execute four or five instructions per clock cycle, thanks to their VLIW architecture. To take into account the possible fill rate of the five (Barts) or four (Cayman) VLIW lanes, we report the best-case and worst-case $\mathtt{CI}_{\mathtt{HW}}$ in Table II. We note that the most recent nVidia architecture is characterized by a relatively low IPS value, when compared with the previous ones from the same producer: This is to be ascribed to the fact that the Kepler architecture SPs are clocked at the same frequency of the whole chip, in contrast with the previous ones, where the execution units ran at twice the chip clock. The last row of Table II reports the value of the $\mathtt{CI}_{\mathtt{HW}}$ for the platforms employed in the exploration. We note that the I/B figures of all the GPUs, except for the Geforce GTX 260, are lower than the one of the CPU: This is in accordance with the purpose of the device, because the GPUs are intended to be a computation accelerator, and are thus meant to be employed in operations intrinsically more demanding than the general purpose load served by the CPU. In current computer architectures, the most limiting bottleneck between the two

Table II. Testbed specifications. RAM clocks are given for the on-device memory when concerning graphics processor units (GPUs) and for the main RAM for the CPU.

	nVidia GPU				AMD GPU		Intel CPU
Architecture	GT200	Fermi	Fermi	Kepler	Barts	Cayman	Sandy Bridge
Series	GeForce	Quadro	GeForce	GeForce	Radeon	Radeon	Core i7
Model	GTX 260	2000M	GTX 470	GTX 660 Ti	HD 6850	HD 6950	2820QM
Cores	192	192	448	1344	960	1280	4 (HT)
Core clock (MHz)	576	550	607	915	775	750	2300–3400
RAM (GiB)	0.875	2.00	1.25	2.00	1.00	1.00	8.00
RAM clock([GHz)	1.00	1.33	1.67	1.50	1.00	1.25	1.33
RAM generation	DDR-3	DDR-3	DDR-3	DDR-5	DDR-5	DDR-5	DDR-3
Peak GIPS	221.18	211.2	543.87	1229.76	744 [5-way]	960 [4-way]	46–68
Bandwidth (GiB/s)	141.7	28.8	133.9	144.2	128	160	25.6
$\mathtt{CI}_{\mathtt{HW}}$ (I/B)	1.45	6.83	3.78	7.94	5.81–27.07	6–23.1	1.79–2.64

All reported clocks are the base clock values. The peak computation power is expressed in billions of *instructions per second* (GIPS); the communication bandwidth with the RAM is expressed in multiples of 2^{30} bytes per second. The last row provides the value of the $\mathtt{CI}_{\mathtt{HW}}$ metric for the platform expressed in I/B.

aforementioned physical limits is the memory bandwidth, because of the ever-growing CPU-memory gap. In this context, we note that nVidia GPU architectures have an I/B index significantly lower than the ones proposed by AMD, in the best lane-filing case, thus hinting at the fact that the latter will perform better when the code being run is strongly computation oriented, with limited memory accesses.

Concerning the software platforms on which the benchmarks were run, AMD GPU binaries were compiled with the AMD APP SDK 2.7, and AMD Catalyst 12.11 provided the drivers and runtime. For the nVidia platforms, we employed the CUDA toolkit 4.2.9 paired with nVidia drivers 313.30, while the CPU OpenCL support was provided by Intel's OpenCL toolchain, build 31360.31426. In order to have a fair evaluation of our code transformations only, we employed the OpenCL compiler at the standard optimization level (-O2).

5.3. Experimental evaluation

To validate the use of the $\mathtt{CI}_{\mathtt{HW}}$ and $\mathtt{CI}_{\mathtt{kernel}}$, we provide an in-depth implementation-space exploration geared at optimizing the implementation of the chosen algorithms applying the techniques described in Section 3.

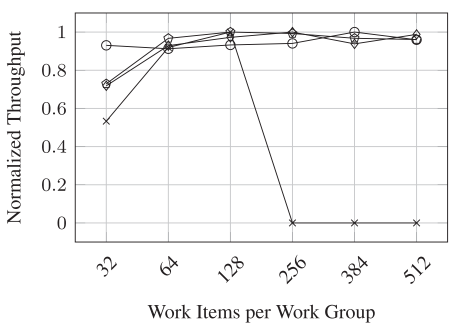
5.3.1. Number of work items per work group. The first optimization choice to be performed when implementing an OpenCL-based application is picking the number of work items per work group. Figure 3 shows the results obtained for the three cryptographic primitives described in Section 4, with the number of work items per work group in the [32–512] range. The choice for the lower bound of the work-items number is justified by both the AMD and nVidia architectures dispatching the threads taking care of computing work items in groups of 32, in turn implying that a work group with less than 32 work items would surely be suboptimal for these architectures, as also confirmed in [9]. In the reported results, some work-items amounts are not allowed to be picked for a single work group, as the underlying hardware does not provide enough resources. These situations, depicted as having zero throughput, are caused by either the lack of enough registers to hold the variables of a single work group (in the Quadro 2000M case) or an explicit limit of 256 work items per work group imposed by the issue unit (in the case of both the AMD architectures). The best throughputs for nVidia architectures are achieved with 384 work items per work group: This can be ascribed to the number being exactly half of the issue queue length of the stream multiprocessors of the underlying hardware and thus allowing the issue units to mask memory latency interleaving instructions coming from two different work items.

The best throughputs for AMD architectures are achieved with 256 work items wide work groups as this is the widest possible size, enabling more freedom in scheduling for the thread-issue units.

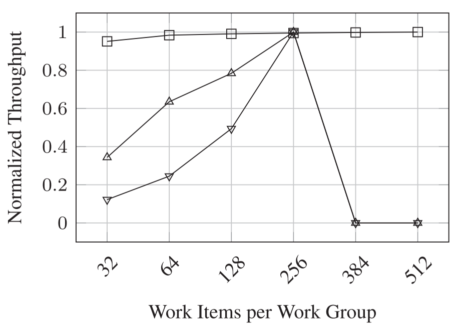
In general, increasing the number of work items per work group effectively yields an increment in the size of the data that is transferred, effectively lowering the $\mathtt{CI}_{\mathtt{kernel}}$ of the implementations. We note that, because increasing the memory bandwidth load does not decrease the performances, the considered implementations have a $\mathtt{CI}_{\mathtt{kernel}}$ higher than the $\mathtt{CI}_{\mathtt{HW}}$ of the underlying architecture. In addition, this is fostered by the ability of the architectures to perform computational operations coming from different work groups during the memory stalls of one, effectively hiding the latencies of the extra memory actions.

5.3.2. Bit slicing. The second implementation choice is whether or not to employ a bit slicing technique to speedup the execution of an algorithm. We recall that this technique allows to quickly modify single bits of the values involved in a computation, at the cost of a rather significant increase in the register pressure. At the same time, each instruction of the implementation will perform in parallel the same operation on as many bits as the architecture word length. All the following implementations employ 32-bit wide variables for bit slicing. Provided there are a large number of operations acting on individual bits; the $\mathtt{CI}_{\mathtt{kernel}}$ of the implementation is lowered because of both the reduced number of instructions and the higher memory pressure.

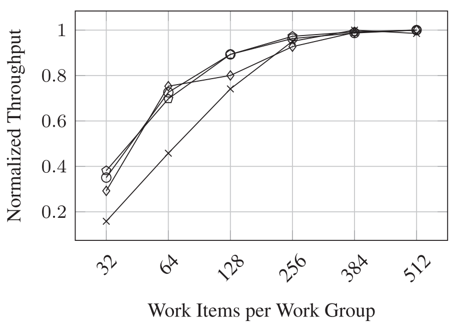
We excluded a priori the use of the bit slicing technique for the `MD5`-crypt algorithm as it would lead to a sure performance drop because of the lack of a significant amount of operations



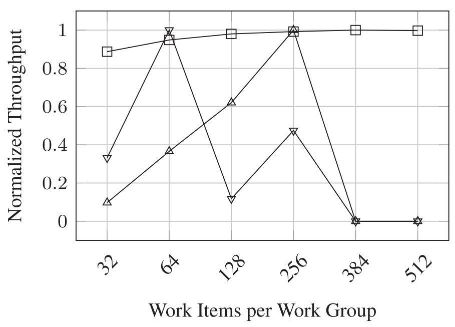
(a) DES-nVidia architectures



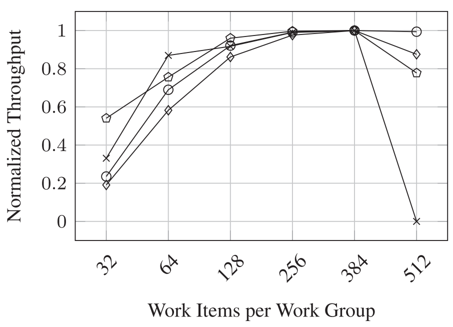
(b) DES-AMD architectures and CPU



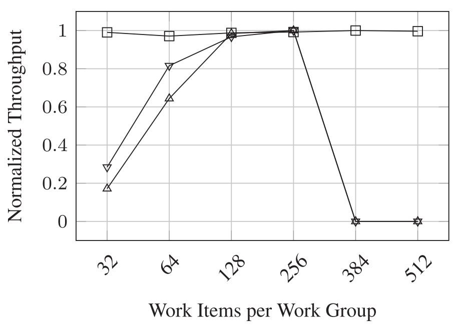
(c) KeeLoq-nVidia architectures



(d) KeeLoq-AMD architectures and CPU



(e) `MD5`-crypt -nVidia architectures

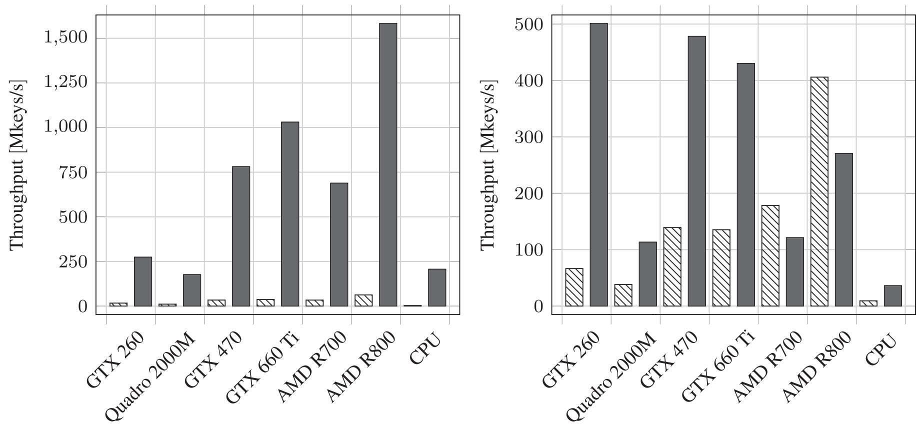


(f) `MD5`-crypt -AMD architectures and CPU

Figure 3. Exploration of performances varying the number of threads per block: The throughput is normalized with respect to the best one achieved on the same hardware. A normalized throughput of zero represents the impossibility of running the algorithm on the architecture with the corresponding number of work items per work group because of lack of resources. nVidia architectures: GTX 260 —×—, Quadro 2000M —⬠—, GTX 470 —◇—, GTX 660 Ti —○— AMD architectures: HD 6850 —▽—, HD 6950 —△— CPU: Intel OpenCL back end —□—.

on individual bits. Indeed, applying a bit slicing technique to `MD5`-crypt would effectively slow down the execution significantly because all the arithmetic operations should be computed bitwise, as described in Section 3. Figure 4 reports the results of applying the bit slice technique to the implementation of DES and KeeLoq on our target platforms. The results show that implementing KeeLoq employing the bit slicing technique yields a significant performance boost: This is justified by both the algorithm being composed exclusively of single-bit operations and the limited size of the memory required by the algorithm, which does not yield an excessive register pressure.

By contrast, the bit sliced implementation of DES does not yield the same benefits for all platforms; this is because DES acts only partly bitwise on its intermediate values, limiting the reduction in the number of instructions; in addition, it has a larger working set when implemented with the bit slicing technique.



(a) KEELOQ Encryption

(b) DES Encryption

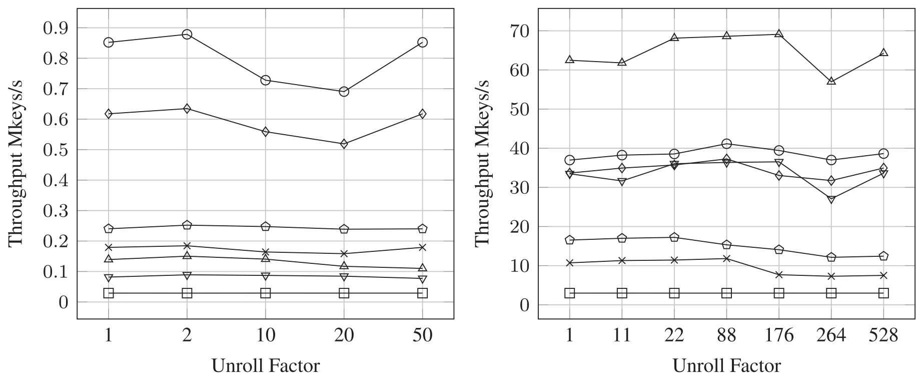
Figure 4. Effects of applying the bit slice technique employing 32-bit wide registers. Hatched lines ▨▨ show the throughput of the vanilla implementations of KEELOQ and Data Encryption Standard (DES) on the considered platforms, while solid columns ▮▮ denote the throughput of the bit sliced ones.

As far as the AMD architectures go, these GPUs take a significant performance hit when running the bit sliced implementation as opposed to the vanilla one. This can be ascribed to them having both an $\texttt{CI}_{\texttt{HW}}$ significantly higher than their nVidia counterparts and to the availability of dedicated hardware instructions for individual bit setting/clearing in the AMD's ISA. Analyzing the assembly code emitted on the AMD platforms, the $\texttt{CI}_{\texttt{kernel}}$ of the vanilla implementations of DES results to be 4.39/4.86 VLIW instructions/Bytes for both Barts and Cayman, respectively. Indeed, these figures are lower than the $\texttt{CI}_{\texttt{HW}}$ lower bounds of both architectures (5.81 for the HD 6850 and 6 for the HD 6950), confirming the memory bound nature of the implementation. On the other hand, the $\texttt{CI}_{\texttt{kernel}}$ corresponding to the bit sliced implementations show a significant drop because of both a massive amount of spills (around 17.4 MiB for the HD 6850 and 12.7 MiB for the HD 6950) and a roughly sixfold increase in the number of instructions. This yields a $\texttt{CI}_{\texttt{kernel}}$ of $\approx 2 \times 10^{-3}$ for the HD 6850 and $\approx 3 \times 10^{-3}$ for the HD 6950, pointing to an implementation so memory intensive that the GPU is not able to mask the stalls caused by memory accesses even through interleaving executions from different work-groups.

Finally, nVidia architectures benefit from the bit slicing of DES because of both the absence of ad-hoc bit mangling instructions and the possibility to employ the stream multiprocessor shared memory as extra registers to reduce the effective bandwidth requirements toward slower memories, helping to compensate the extra register pressure. A notable result is that the performance of bit sliced DES do not improve when moving to more recent architectures in the nVidia group. This counterintuitive result can be justified examining the $\texttt{CI}_{\texttt{HW}}$ of the the three desktop GPUs by nVidia: The oldest one is the one actually having the lowest $\texttt{CI}_{\texttt{HW}}$ and thus expected to be able to cope better with an optimization that increases the required memory while lowering the number of instructions.

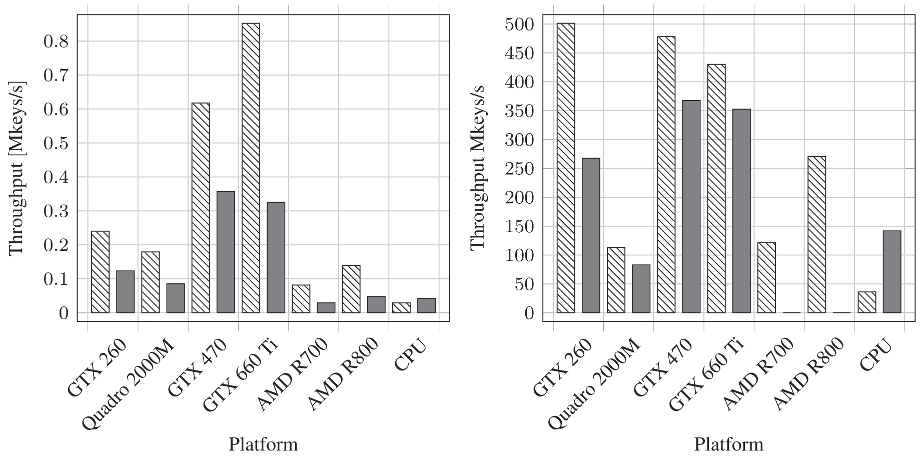
5.3.3. Loop unrolling. After considering the use of the bit slicing technique, we analyzed the effects of unrolling partially the loops involved in our algorithm. The DES algorithm was always run as fully unrolled, as the number of iterations of the loop is low (16), and we observed that no advantages were gained from executing it without a full unroll. On the other hand, we explored the feasibility of unrolling a variable amount of the iterations of the `MD5`-crypt and KEELOQ main loops.

Figure 5 reports the effects of unrolling up to 50 executions of the 1000 iterations loop of `MD5`-crypt, because the compiler refused to unroll a larger number of iterations, and up to a full unroll of KEELOQ main loop. The reported results show that unrolling iterations of the `MD5`-crypt yields small performance increases for low unrolling factors and results in a noticeable performance drop if the loop is unrolled more than 10 times. This can be ascribed to the large code size of the `MD5`-crypt



(a) `MD5`-crypt (b) KEELOQ-BS

Figure 5. Effects of applying loop unrolling. The compiler refuses to unroll 50 or more iterations of the `MD5`-crypt main loop, while a full KEELOQ unroll was possible. nVidia architectures: GTX 260 —×—, Quadro 2000M —⬠—, GTX 470 —◇—, GTX 660 Ti —○—; AMD architectures: HD 6850 —▽—: HD 6950 —△—; CPU: Intel OpenCL back end —□—.



(a) `MD5`-crypt (b) DES Encryption

Figure 6. Effects of employing vector types: with vector types ▮▮, without them ▨▨. Using vector types yields an effective benefit only in the case of Intel's CPU OpenCL back end. The zero result for AMD platforms running Data Encryption Standard (DES) is caused by the compiler exhausting the system resources trying to compile the kernel.

loop body, which actually encompasses an entire `MD5` hash: The advantage of unrolling such a large loop are limited as the time overhead saved from not computing branches is small with respect to the loop body. By contrast, performing loop unrolling on KEELOQ yields reasonable performance increases, especially on the GPU platforms endowed with an instruction cache (AMD ones and Kepler). The performance advantage is greater on AMD GPUs, provided the unrolling is limited to a quarter of the loop executions at most. The performance loss with greater unroll amounts is due to the code no longer fitting the instruction cache of these platforms.

5.3.4. Vector types. The use of vector types, strongly encouraged by OpenCL programming manuals, provides the compiler with an explicit indication of data parallelism in the computation. Employing vector types in an implementation lowers the $\mathtt{CI}_{\mathtt{kernel}}$ as the quantity of data to be processed is fourfold, while the number of instructions does not increase if the architecture is endowed with vector units. Figure 6 shows how both nVidia and AMD platforms do not improve their performance when employing vector integers in the implementation. This is to be ascribed to the increased register pressure imposed by the vector integers, which in turn causes spills to the memory effectively lowering significantly the $\texttt{CI}_{\texttt{kernel}}$ of the emitted code, up to a point where the computation is memory bound. As an example, we report the $\texttt{CI}_{\texttt{kernel}}$ of the HD 6850 implementation of the MD5-crypt, which are respectively 646 without using vector integers and 0.036 employing them. The massive drop in the $\texttt{CI}_{\texttt{kernel}}$ is caused by the extra read/write operations inserted by the compiler to cope with the register pressure, resulting in a significant amount of memory transfer actions (in the MB range) both toward the shared scratchpad and global memory. By contrast, the nonvectorized implementation has a negligible amount of memory transfers per kernel execution. We also note that the compiler was not able to allocate the resources for the AMD platform implementation of the DES algorithm exploiting vector integers, hence the depicted zero throughput. Moreover, we note also that employing vector types with KeeLoq shows the same behavior of DES, that is, a performance drop, because of the rather significant register pressure imposed by the bit slicing. The only platform that benefits from the use of vector integers is the CPU-based OpenCL back end by Intel, which, we note, makes effective use of the CPU vector instructions to achieve a consistent speedup. We note that the large amount of cache memory available on the CPU is able to effectively mitigate the extra memory access requirements, given the high spatial and temporal locality of the data.

5.3.5. Summary. After performing our implementation space exploration, we now provide the results concerning the combinations of optimizations that we have found to be the best performing ones on the target platforms.

Table III reports the maximum achieved throughputs for each platform, including both the regular implementations and the one employing the bit slice technique. Table IV reports the computational index of the vanilla implementations as different from the bit sliced ones because of the significant changes that the optimizations bring to the corresponding $\texttt{CI}_{\texttt{kernel}}$.

Table III. Perfomance comparison: maximum achieved throughputs [Mkeys/s] (MD5-crypt in [kkeys/s]) for each platform.

	nVidia GPU				AMD GPU		Intel CPU
Series Model	GeForce GTX 260	Quadro 2000M	GeForce GTX 470	GeForce GTX 660 Ti	Radeon HD 6850	Radeon HD 6950	Core i7
DES	66.44	38.17	139.44	135.35	178.3	405.86	9.25
DES-BS	501.16	113.35	478.15	430.08	121.2	270.4	36.13
KEELOQ	17.22	11.81	37.26	41.14	36.5	69.1	2.98
KEELOQ-BS	273.93	176.10	780.93	1030.69	688.4	1583.4	206.25
MD5-crypt	252.18	184.44	634.75	878.51	89.1	150.2	42.19

GPU, graphics processor unit; DES, Data Encryption Standard.

Table IV. $\texttt{CI}_{\texttt{kernel}}$ of the algorithm implementations compared against the $\texttt{CI}_{\texttt{HW}}$ of the underlying platforms.

	nVidia GPU				AMD GPU	
Series Model	GeForce GTX 260	Quadro 2000M	GeForce GTX 470	GeForce GTX 660 Ti	Radeon HD 6850	Radeon HD 6950
Platform $\texttt{CI}_{\texttt{HW}}$ [I/B]	1.45	6.83	3.78	7.94	5.87–27.07	6–23.10
DES	14.09	3.79	4.00	3.45	4.39	4.89
DES-BS	22.09	1.46	1.53	1.47	2×10^{-3}	3×10^{-3}
KEELOQ	3716.6	63.56	80.03	63.58	29.10	14.7
KEELOQ-BS	120.58	6.83	6.77	6.70	5×10^{-3}	0.416
MD5-crypt	6.43	3.68	3.68	3.68	646.06	655.94

GPU, graphics processor unit; DES, Data Encryption Standard.

In the case of the DES cipher, the straightforward implementation turns out to be more computationally intensive (i.e., has an higher $\mathtt{CI}_{\mathtt{kernel}}$) than the optimal point of the GT200 and mobile Fermi devices, while being almost on par with the GTX 470 discrete Fermi card. As a consequence, the throughputs are proportional to the effective amount of IPS, the aforementioned platforms. By contrast, the GeForce GTX 660 card is limited by its memory bandwidth, a fact confirmed by its DES throughput being roughly the same of the GTX 470, which has the same memory bandwidth. Concerning the AMD architectures, the $\mathtt{CI}_{\mathtt{kernel}}$ computed considering VLIW instruction bundles shows that the DES is to be considered slightly memory bound on both of them. The reason for the higher throughput achieved by the HD 6950 over the HD 6850 is the different fitness of the underlying ISA to DES instructions, which forces the older architecture to a worse fill rate of the VLIW lanes.

The bit sliced implementation of DES on nVidia platforms turns out to be memory intensive for most platforms, in turn resulting in a counterintuitive performance trend, where the newest platform perform worse than the oldest one (Geforce GTX 260). We note that the high $\mathtt{CI}_{\mathtt{kernel}}$ reported for the Geforce GTX 260 implementation is caused by a significantly shorter assembly produced by the compiler for the architecture in exam.

In particular, the register allocator of the nVidia compiler is remarkably efficient on the GT200 architecture, resulting in the GTX 260 assembly being free from memory operations because of spills leading to a significantly lower $\mathtt{CI}_{\mathtt{kernel}}$ wrought the other nVidia GPUs. As far as AMD platforms go, the higher memory pressure imposed by bit sliced DES is not managed properly and causes a massive amount of memory spills, effectively yielding low throughputs, with respect to the nVidia counterparts. The KEELOQ cipher, in the straightforward implementation is characterized by a high $\mathtt{CI}_{\mathtt{kernel}}$, with respect to the ones offered by the platforms, yielding performances bound only by their computational capabilities. The bit sliced implementation of the same cipher, as expected, yields lower $\mathtt{CI}_{\mathtt{kernel}}$ values for all the platforms, because of issues in coping with the increased register pressure. However, the $\mathtt{CI}_{\mathtt{kernel}}$ figures for all the nVidia platforms are still above the corresponding $\mathtt{CI}_{\mathtt{hw}}$, save for the GT 660 Ti. It is possible to notice that the algorithm speedups are in fact bound to the GPU computational power of the nVidia GPUs, save for the Kepler board that underperforms with respect to a purely computational boost expectation. AMD architectures are able to cope worse with the strong register pressure imposed by KeeLoq bit slicing, and the corresponding compiler outputs an assembly dense with spill actions. The consequence is a very low $\mathtt{CI}_{\mathtt{kernel}}$, which binds their performances mostly to memory throughput. The reason for the performance increase between the nonbit sliced and bit sliced version of KeeLoq on AMD architectures is the implicit high parallelism offered by a bit sliced implementation that, in the case of KeeLoq, computes almost $32\times$ the amount of encryptions with the same amount of instructions of a nonbit sliced one, even taking into account the required encoding and decoding.

Finally, In the case of the MD5-crypt primitive, the optimal implementation for nVidia architectures has substantially the same $\mathtt{CI}_{\mathtt{kernel}}$ for all of them, save for the GTX 260, where the compiler manages a higher $\mathtt{CI}_{\mathtt{kernel}}$ avoding memory spills. The resulting $\mathtt{CI}_{\mathtt{kernel}}$ figures imply that the algorithm is memory on all the nVidia devices save the GTX 260, with the GTX 470 coming relatively close to be exactly balanced in computation and memory requirements. The performance figures reflect this trend, with the Kepler board delivering only 30% more througput of the Fermi, despite a 130% advantage in computing power.

AMD architectures show a significantly high $\mathtt{CI}_{\mathtt{kernel}}$ for the best performing MD5-crypt implementation on them, pointing to a strong computational bound on their performances. This in turn results in the newer architecture performing better on the MD5-crypt kernel, although a significant portion of the improvement is provided by the addition of native integer operations (which are common in MD5-crypt) on it.

Figure 7 summarizes the degree of fitness of the examined architectures in terms of number of computed keys per second, divided by the computational capability of the platform expressed in IPS. The resulting figure of merit (computed keys per instruction) is normalized by the best value achieved for that algorithm, to ease comparisons. We note that the best architecture exploitation are achieved on the platforms where the $\mathtt{CI}_{\mathtt{kernel}}$ of the implementation is closer to the

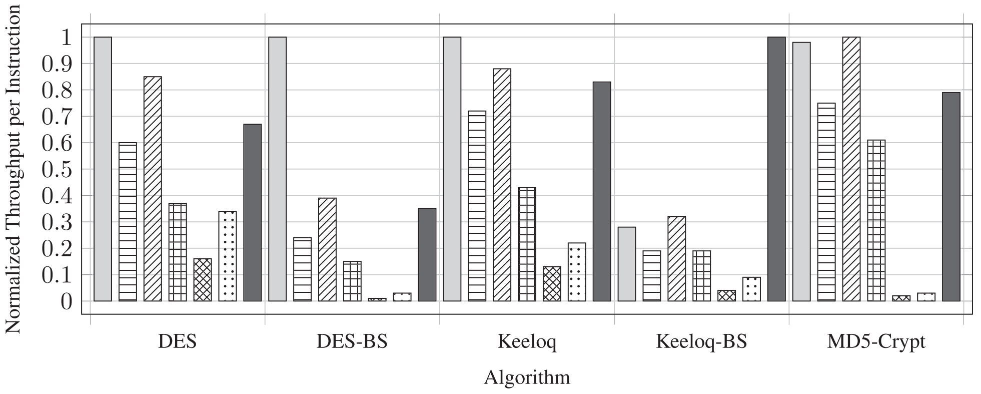


Figure 7. Computed keys per instruction, normalized by the fastest architecture throughput for each cipher. nVidia architectures: GTX 260 ▯, Quadro 2000M ▤, GTX 470 ▨, GTX 660 Ti ▦; AMD architectures: HD 6850 ▩: HD 6950 ⋮; CPU: Intel OpenCL back end ■.

one of the underlying hardware, without going below it. In particular, the GTX 260 obtains a good exploitation level over the board as the $\mathtt{CI}_{\mathtt{HW}}$ of the hardware is particularly low. The only case where a comparable result is achieved is the one of the `MD5`-crypt, where the GTX 470 has an $\mathtt{CI}_{\mathtt{HW}}$ figure closer to the one of the algorithm. The results show how the AMD platforms do not perform well with respect to their maximum computing power: This can be ascribed to their comparatively high hardware $\mathtt{CI}_{\mathtt{HW}}$, which makes them run as memory bound in most of the cases.

The $\mathtt{CI}_{\mathtt{kernel}}$ computed on the implementation is a good indicator of the degree of fitness of a computation to a target platform and is thus viable to predict the performances at compile time. This in turn represents an enabling strategy to evaluate the level of performance portability of algorithms across different architectures, driving choices such as which optimizations should be employed. In particular, we note that computing the $\mathtt{CI}_{\mathtt{HW}}$ for server class hardware of the same architecture (Tesla boards for nVidia, FirePro for AMD, and a Xeon E5-2687W for CPUs) yields slightly decreased but coherent $\mathtt{CI}_{\mathtt{HW}}$ values. In particular nVidia Tesla C1060 (GT200), C2075 (Fermi), and K40C (Kepler) boards have an $\mathtt{CI}_{\mathtt{HW}}$ of 3.04, 3.57, and 7.45, respectively, while the AMD FirePro V7900 (Cayman) is characterized by a $\mathtt{CI}_{\mathtt{HW}}$ of 23.2 (no server-grade equipment with the Barts architecture has been produced). These figures are closer to the desktop-grade board and generally slightly lower, because of the use of better memories (able to work at higher clock rates), and wider memory busses on server-grade equipment. This trend is confirmed comparing the $\mathtt{CI}_{\mathtt{HW}}$ indexes for the Xeon E5-2687W (0.48, 0.59 when employing turbo mode) where the differences from the laptop CPU are more evident because of the presence of a double memory bus on the Xeon CPU, resulting in a significant reduction in the $\mathtt{CI}_{\mathtt{HW}}$. Following this trend, we expect server grade equipment to be slightly more tolerant with respect to register spill and extra memory actions than its desktop counterpart.

To conclude our exploration, we provide a high level summary of the effects of the transformations on the performances obtainable on the explored platforms in Table V.

5.4. Brute force-resistant cryptographic hashing

Having chosen cipher key and password brute forcing as the application domain of our investigation, we now provide some guidelines to design brute forcing-resistant password hashing techniques. We note that, to hinder fast execution on our target platforms, it is helpful for an algorithm to present a very low I/B index so that the maximum achievable throughput in terms of tested keys per second is bound by the bandwidth of the platform on which it is running. We note that, to the end of counteracting their beneficial effect, it should be possible for the brute force-resistant password hashing algorithm to tune the amount of memory so that the required amount of memory will exceed the size of the caches. This approach is rather common and was only lately super-

Table V. Summary of the effects of the different code transformations on the examined platforms: A ✓points to performance improvements employing the technique, while a ✗denotes a decrease in performance.

Transformation	Architecture	DES	KeeLoq	MD5-Crypt
Tuned number of work items per work group	nVidia	✓	✓	✓
	AMD	✓	✓	✓
	CPU	✓	✓	✓
Bit slicing	nVidia	✓	✓	✗
	AMD	✗	✓	✗
	CPU	✓	✓	✗
Loop unrolling	nVidia	✗	✓	✓
	AMD	✗	✓	✓
	CPU	✗	✓	✓
Vector types	nVidia	✗	✗	✗
	AMD	✗	✗	✗
	CPU	✓	✓	✓

DES, DES, Data Encryption Standard.

seded by the proposal of the `scrypt` password hashing primitive, now an Internet Engineering Task Force draft [31]. The `scrypt` primitive tackles the hardening versus brute forcing efforts proposing the concept of a *memory hard* algorithm, that is, an algorithm with asymptotic space complexity equal to the asymptotic time complexity. The `scrypt` primitive employs a common cryptographic hash to build a memory hard password hashing algorithm, which has its access pattern driven pseudorandomly by the input values, which in turn hinders the ability of the cache prefetchers, if they are in place. The aforementioned password hashing algorithm is also proven to expose only a constant, designer tunable, factor of internal parallelism, of the pipeline kind, and no data parallelism, except for the one achieved by running multiple instances of it on different data. The `scrypt` primitive exposes to the designer two choices: an integer work factor, which determines the length of the loops run by the algorithm and the amount of employed memory, and the choice of the cryptographic hash to be employed. In the light of the results obtained, we advise the designers to pick a combination of the aforementioned design criteria that results in very low I/B index so to prevent the efficient parallelization of the brute forcing efforts on GPU platforms. Given the examined architectures, I/B indexes in the range of [0.1–0.4] seem a safe choice, because, regardless of the architectural evolution of the platforms, none of the reported ones offers such a low hardware I/B.

6. ANALYSIS OF THE RELATED WORK VIA `CI` INDEXES

In this section, we provide a survey of the literature works regarding the acceleration of cryptographic algorithms on GPGPUs. Willing to analyze the results reported in Table VI according to their $\mathtt{CI}_{\mathtt{kernel}}$ index, we would need the source code employed to obtain their performance figures. As this is not possible, thus, we cannot directly compute $\mathtt{CI}_{\mathtt{kernel}}$; we will consider the related figure given by the ratio between the peak IPS figure of the platform and the effective throughput of the implementation (`instr/X`), which is an upper bound in all the cases where the algorithm is computationally bound. One of the prime targets for acceleration via GPGPUs has been the AES block cipher: Table 6 reports the most relevant implementations of AES-128 on various GPGPU architectures. Among these implementations, Manavski [10] was the first to provide such an implementation on a CUDA-enabled platform, while several other works provided improved performance using different GPUs, as well as various performance analyses and fine tuning of opti-

Table VI. Comparison of graphics processor unit implementations of Advanced Encryption Standard.

Related work	Platform	Language	Throughput (GB/s)	`instr/X` (I/B)	$\mathtt{CI}_{\mathtt{HW}}$ (I/B)	Mode
[10]	Geforce 8800 GTX	CUDA	1.03	1004.85	11.97	ECB
[9]	Geforce 8400 GS	CUDA	0.11	130.09	2.25	CTR
	Geforce 8800 GT	CUDA	1.56	86.15	2.30	
[32, 33]	Geforce GTX 285	CUDA	4.40	70.68	1.95	ECB
[34]	Geforce 9200M GS	CUDA	0.80	11.00	1.60	ECB
[35, 36]	Geforce 9500 GT	CUDA	0.72	48.88	1.37	CTR
[37]	Geforce GTX 295	CUDA	3.70	149.18	2.47	ECB
		OpenCL	3.17	174.13		
[38]	Tesla C2050	CUDA	6.07	169.68	7.15	ECB
[39]	Geforce GTX 285	C/CUDA	6.20	50.16	1.95	ECB
[40]	Tesla C2050	CUDA	7.50	137.33	7.15	ECB
[41]	2×Xeon X5650	AES-NI	14.84	0.37	0.99	ECB

CUDA, Compute Unified Device Architecture; AES, Advanced Encryption Standard.

mized code (Table 6), topping slightly less than 6.2 GiB/s for Geforce GTX285 and 7.5 GiB/s for Tesla C2050 architectures. Such analyses deal primarily with the definition of the optimal granularity of parallelism, block size, and management of the memory hierarchy (what to allocate to shared memory and how to avoid bank conflicts) to improve the performance of AES. Other works deal with implementations of AES for specialized applications [42, 43] or with automated CUDA kernel generation from `C`-code [39]. However, the introduction of the AES-NI [44] instruction set extension in the Intel Nehalem processor family provides a fast way to implement AES. Niewiadomska *et al.* [41] provide experimental results on Xeon X5650 servers, reaching a throughput slightly less than 15 Gbps, which is in line with the forecast that can be inferred from the Intel white paper [44]. The analysis of the `instr/X` index of the pioneering implementations shows their lack of optimization [10], as significantly lower $\mathtt{CI}_{\mathtt{kernel}}$ figures have been obtained by their successors. It is interesting to note how an implementation on a GPU targeting the mobile market [34], while not achieving especially good results in absolute terms, has the lowest `instr/X` measure, demonstrating the ability to fully exploit the computational resources of the underlying hardware.

Because of the large popularity of DES and its vulnerability to brute-forcing attacks (as it employs only 56 bits of secret key), several research groups have implemented the DES cipher on GPGPU. Table VII reports a summary of the open literature results on this algorithm. Note that in [46], while the authors claim an improvement of a factor of 2 over [45] because of computing bit slices on the fly, the comparison of the `instr/X` figures shows how the effective improvement in the implementation is around 25%. The remaining portion of the reported speedup is to be attributed to the difference in raw computational power between the different GPU platforms used in the experiments. On the other hand, the best speedup is obtained in [37]. The implementation used demonstrates a good value of the $\mathtt{instr/X} < z$ metric, compared with the previous work and to our current implementation. However, it is difficult to establish exactly the reason of this speedup, as the authors report having used several techniques, including code specialization by mode of operation and several tricks to circumvent issues in the compiler (e.g., suboptimal management of memory load operations), although they do not specify which optimizations are actually used in the case of DES. Optimization of memory load coalescing, though, is compatible with the results reported in their work.

Table VII. Comparison of graphics processor unit implementations of Advanced Encryption Standard.

Related work	Platform	Language	Throughput (GB/s)	`instr/X` (I/B)	Comments
[45]	Geforce GTX260	CUDA	0.6	368.6	nonbit sliced, ECB mode
			2.98	74.22	bit sliced, ECB mode
[46]	Geforce GTX275	CUDA	5.54	60.64	bit sliced, ECB mode
[37]	Geforce GTX295	CUDA	15.46	35.76	not bit sliced, ECB mode
		OpenCL	15.48	35.76	not bit sliced, ECB mode
[41]	FirePro V7800	OpenCL	0.66	1527.27	not bit sliced, CFB mode
	Tesla M2050		1.03	66.56	
	Radeon 6790		1.29	520.93	
This work	Geforce GTX260	OpenCL	0.531	416.12	nonbit sliced, ECB mode
			4.00	55.16	bit sliced, ECB mode
This work	Radeon 6950	OpenCL	3.24	1182.67	nonbit sliced, ECB mode
			2.16	1175.14	bit sliced, ECB mode

CUDA, Compute Unified Device Architecture.

Finally, in [47], a first study of the bit-level parallelism of the KEELOQ cipher is reported. With respect to that, we improved the implementation of the nonlinear function of the cipher through adapting the bit slice technique also for it, thus reaching a further $\times 40$ speedup for the GTX260 and a $\times 20$ speedup for the GTX470.

7. CONCLUDING REMARKS

In this paper, we studied the effect of compiler transformations and algorithmic choices on representative applications from the cryptography domain. To provide a useful comparison, we map these applications on four nVidia GPGPUs, two AMD GPGPUs, and on an Intel Sandy Bridge i7 CPU. We introduced the `CI` metric to asses how well an implementation fit a generic architecture. The defined metric, besides being useful to compare either different implementation or algorithmic choices and their fitness to a specific architecture, can also be useful to the compiler to guide the code optimization process through a characterization of the available optimization passes in terms of their impact on the metric. We foresee, as two possible interesting directions for future research, two specializations of the `CI` index. The first one is aimed at tackling a structured memory hierarchy, such as the one of OpenCL local and global memories, weighing the information transfer from faster memories less than a load/store action from slower ones. A second possible specialization could include the inter-CPU transfers in SMP systems, taking into account thus the capability of the platform to bypass the loads/stores into main memory through the use of shared caches.

APPENDIX A: KEELOQ IMPLEMENTATIONS

In this section, we will provide the code samples of the KeeLoq implementations for both the plain and bit sliced ones.

A.1. Plain implementation

The plain OpenCL implementation of KeeLoq is taken from the reference algorithm code, modifying it to exploit the fact that, during brute forcing, only a single valid key will be found. To this end, the following code computes a KeeLoq encryption per work item, generating it exploiting the global work item id. At the end of the computation, the work item writes a specific global memory slot with the result of the comparison of the correct decryption of the ciphertext: This practically performs not only the brute forcing but also the correctness check for the key in parallel. The unrolling factor is driven via a `#pragma` suggestion to the compiler.

```
#define KeeLoq_NLF    0x3A5C742E
#define bit(x,n)     (((x)>>(n))&1)
#define g5(x,a,b,c,d,e) (((x>>a)&1)|((x>>(b-1))&2)|((x>>(c-2))&4)|((x>>(d-3))&8)
| ((x>>(e-4))&16))

__kernel void keeloq(uint plaintext, uint ciphertext,
                     ulong key, __global uint* results)
{
  uint global_id = get_global_id(0);
  key += global_id;
#if UNROLL > 1
  #pragma unroll UNROLL_FACTOR
#endif
    for (uint r = 0; r < 528; r++)
    {
      plaintext = (plaintext >> 1) ^
            ((bit(plaintext, 0) ^ bit(plaintext, 16) ^
            ((uint) bit(key, r & 63)) ^
            bit(KeeLoq_NLF, g5(plaintext, 1, 9, 20, 26, 31))) << 31);
    }
  *(results + global_id) = (plaintext != ciphertext);
}
```

A.2. Bit sliced implementation

We report hereafter the details of the bit sliced KeeLoq implementation. Because bit sliced implementations code is typically highly redundant, it is commonplace to generate it, instead of implementing it by hand. We followed this same approach, and we will thus now report the code stub into which the core of the bit sliced KeeLoq is generated. The first step to implement KeeLoq in a bit sliced fashion is to have a computational implementation of the nonlinear function (NLF), acting on register-sized variables instead of single bits, as follows:

```
uint nlf(uint a, uint b, uint c, uint d, uint e)
{
    return b^a^(e&c)^(e&a)^(d&c)^(d&a)^(c&b)^(b&a)^(e&b&a)^(e&c&a)^(e&d&b)^(e&d&c);
}
```

The core of the bit sliced KeeLoq is represented by the actual OpenCL kernel computing the cipher, of which we now report a version shortened in the points where our code was automatically generated, to the end of providing a high level view.

```
#define bit(x,n)  (((x) >> (n)) & 1)

__kernel void keeloq(uint plaintext, uint ciphertext,
                     ulong key, __global uint* result)
{
    uint global_id = get_global_id(0);
    key += global_id * KEYS_PER_KERNEL;
    uint result_mask = 0xFFFFFFFF;

    /* Initialization of the key LSBs */
    uint key00 = 0x55555555;
    uint key01 = 0x33333333;
    uint key02 = 0x0F0F0F0F;
    uint key03 = 0x00FF00FF;
    uint key04 = 0x0000FFFF;

    /* Initialization of the key MSBs */
    uint key05 = bit(key, 5) ? (uint) -1 : 0;
    ... /* more generated lines */
    /* Encoding of the plaintext */
    uint plaintext00 = bit(plaintext, 0) ? (uint) -1 : 0;
    ... /* more generated lines */

    /* Encoding of the ciphertext */
    uint ciphertext00 = bit(ciphertext, 0) ? (uint) -1 : 0;
    ... /* more generated lines */

    /* Cipher computation */
    plaintext00 = plaintext00 ^ plaintext16 ^ key00 ^
                  nlf(plaintext01, plaintext09, plaintext20,
                      plaintext26, plaintext31);
    ... /* more generated lines */

    /* Final comparison */
    result_mask &= ~(plaintext00 ^ ciphertext16);
    ... /* more generated lines */
    result[global_id] = result_mask;
}
```

The KeeLoq implementation starts through gathering the work-item ID through the `get_global_id` OpenCL call and computes the value of the first key that should be brute forced by the kernel. The 1-bit results of the brute forcing effort, contained in the `result_mask` variable, are initialized to 1. The bit sliced encoding of the five least significant bits of the 32 keys to be tried in a single bit sliced KeeLoq execution are the same throughout all the execution and are thus initialized via compile-time known constants. By contrast, the rest of the actual key bits are to be computed at each time and will not change over the 32 keys being computed: They are thus encoded in the `key05`–`key63` integer values, filling them with either all ones or all zeroes. To this end, the key encoding lines can be generated by the following Python snippet:

```
for i in xrange(5,64):
  print "uint␣key%s␣=␣bit(key,␣%d)␣?␣(uint)␣-1␣:␣0;" % (i, i)
```

Similarly, all the input plaintext and ciphertext bits are encoded in different integer variables, and the required code can be generated as follows:

```
for i in xrange(32):
  print "uint␣plaintext%s␣=␣bit(plaintext,␣%d)␣?␣(uint)␣-1␣:␣0;" % (i, i)

for i in xrange(32):
  print "uint␣ciphertext%s␣=␣bit(ciphertext,␣%d)␣?␣(uint)␣-1␣:␣0;" % (i, i)
```

The round function of KeeLoq is then computed, acting directly on register sized variables, performing the state update registerwise and employing register renaming to minimize the amount of register to register `move` instructions. The code of the 528 round computations can be generated as follows:

```
for i in xrange(528):
  print "plaintext{destination}␣=␣plaintext{zero}␣^␣plaintext{sixteen}␣^" \
      "␣key{key_index}␣^␣nlf(plaintext{one},␣plaintext{nine},␣plaintext{twenty}," \
      "␣plaintext{twentysix},␣plaintext{thirtyone});" \
      .format(destination=(i%32)), zero=(i%32), one=((i+1)%32), nine=((i+9)%32),
      sixteen=((i+16)%32), twenty=((i+20)%32), twentysix=((i+26)%32),
      thirtyone=((i+31)%32), key_index=str(i&63).rjust(2, "0"))
```

The bit sliced check for the correctness of the decrypted ciphertext are performed, setting to 1 the bit of the `result_mask` value that corresponds to the instance, out of the 32 computed in parallel, which has found the correct key. The code for the final check is generated as:

```
for i in xrange(32):
  print "result_mask␣&=␣~(plaintext%s␣^␣ciphertext%s);" % (i, ((i + 16) % 32))
```

Finally, the value in `result_mask` is saved to memory, where it is retrieved by the host.

REFERENCES

1. Duranton M, Black-Schaffer D, Bosschere KD, Maebe J. The HiPEAC vision for advanced computing in horizon 2020. *Technical Report*, HiPEAC Network of Excellence, 2013.
2. Khronos WG. OpenCL– the open standard for parallel programming of heterogeneous systems, 2011. (Available from: http://www.khronos.org/opencl/) [Accessed on 21 March 2014].
3. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell T. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 2007; **26**(1):80–113.
4. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU Computing. *Proceedings of the IEEE* 2008; **96**(5):879–899.
5. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-purpose GPU Programming* (1st edn). Addison-Wesley Professional: Indianapolis, Indiana, 2010.
6. Bayoumi AM, Chu M, Hanafy YY, Harrell P, Refai-Ahmed G. Scientific and engineering computing using ATI stream technology. *Computing in Science and Engineering* 2009; **11**(6):92–97.
7. CAPS Enterprise. OpenHMPP ver. 2.5, 2013. (Available from: http://www.caps-entreprise.com/openhmpp-directives) [Accessed on 21 March 2014].
8. The OpenACC™ Programming Interface, ver. 1.0, 2011. (Available from: http://www.openacc-standard.org/) [Accessed on 21 March 2014].
9. Di Biagio A, Barenghi A, Agosta G, Pelosi G. Design of a parallel AES for graphics hardware using the CUDA framework. *IPDPS*, IEEE, Rome, Italy, 2009; 1–8.
10. Manavski SA. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. *IEEE International Conference on Signal Processing and Communication (ICSPC 2007)*, Dubai, United Arab Emirates, 2007; 65–68.
11. Harrison O, Waldron J. AES encryption implementation and analysis on commodity graphics processing units. In *CHES*, vol. 4727, Paillier P, Verbauwhede I (eds)., LNCS. Springer: Berlin Heidelberg, 2007; 209–226.
12. Agosta G, Barenghi A, De Santis F, Di Biagio A, Pelosi G. Fast disk encryption through GPGPU acceleration. *PDCAT*, IEEE Computer Society, Higashi, Hiroshima, Japan, 2009; 102–109.
13. Fung WWL, Sham I, Yuan GL, Aamodt TM. Dynamic warp formation and scheduling for efficient GPU control flow. *MICRO*, IEEE Computer Society, Chicago, Illinois, USA, 2007; 407–420.
14. Garland M, Grand SL, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V. Parallel computing experiences with CUDA. *IEEE Micro* 2008; **28**(4):13–27.
15. Advanced Micro Devices Inc. R800 Evergreen family instruction set architecture, 2013. (Available from: http://developer.amd.com/wordpress/media/2012/10/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf) [Accessed on 21 March 2014].
16. Advanced Micro Devices Inc. R700 family instruction set architecture specifications, 2013. (Available from: http://developer.amd.com/wordpress/media/2012/10/R700-Family_Instruction_Set_Architecture.pdf) [Accessed on 21 March 2014].
17. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *ACM Queue* 2008; **6**(2): 40–53.
18. nVidia Corp. CUDA technology, 2008. (Available from: http://www.nvidia.com/CUDA) [Accessed on 21 March 2014].
19. nVidia Corp. Geforce GTX 260 specifications, 2013. (Available from: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-260/specifications) [Accessed on 21 March 2014].
20. nVidia Corp. Fermi architecture whitepaper, 2013. (Available from: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf) [Accessed on 21 March 2014].
21. nVidia Corp. Kepler architecture whitepaper, 2013. (Available from: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf) [Accessed on 21 March 2014].
22. Kirk DB, Hwu WW. *Programming Massively Parallel Processors: A Hands-on Approach*, Applications of GPU Computing Series. Elsevier Science, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2010. (Available from: http://books.google.it/books?id=qW1mncii_6EC) [Accessed on 21 March 2014].
23. Bacon DF, Graham SL, Sharp OJ. Compiler transformations for high-performance computing. *ACM Computing Surveys* 1994; **26**(4):345–420.
24. Biham E. A fast new DES implementation in software. In *FSE*, vol. 1267, Biham E (ed.)., LNCS. Springer, Berlin Heidelberg, 1997; 260–272.
25. Harrison O, Waldron J. Practical symmetric key cryptography on modern graphics hardware. In *Usenix Security Symposium*, van Oorschot PC (ed.). USENIX Association, Berkeley, California, USA, 2008; 195–210.
26. Advanced Micro Devices Inc. OpenCL Programming Guide for APP Platforms, 2013. (Available from: http://developer.amd.com/wordpress/media/2012/10/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf) [Accessed on 21 March 2014].
27. NIST. FIPS-46-3: Data Encryption Standard (DES), 1999. (Available from: http://www.itl.nist.gov/fipspubs/) [Accessed on 21 March 2014].
28. Microchip. KeeLoq™ technology, 2013. (Available from: http://www.microchip.com/pagehandler/en-us/technology/embeddedsecurity/technology/home.html) [Accessed on 21 March 2014].
29. Poul-Henning K. md5crypt(): implementation of the MD5 one-way hashing algorithm, 1995. (Available from: http://phk.freebsd.dk/sagas/md5crypt_eol.html) [Accessed on 21 March 2014].

30. Rivest R. The MD5 message-digest algorithm, 1992. RFC Editor.
31. Colin P. The scrypt password-based key derivation function, 2013. (Available from: https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-01) [Accessed on 21 March 2014].
32. Iwai K, Kurokawa T, Nishikawa N. AES encryption implementation on CUDA GPU and its analysis. *ICNC*, IEEE Computer Society, Higashi Hiroshima, Japan, 2010; 209–214.
33. Iwai K, Nishikawa N, Kurokawa T. Acceleration of AES encryption on CUDA GPU. *IJNC* 2012; **2**(1):131–145.
34. Mei C, Jiang H, Jenness J. CUDA-based AES parallelization with fine-tuned GPU memory utilization. *IPDPS Workshops*, IEEE, Atlanta, Georgia, USA, 2010; 1–7.
35. Tran NP, Lee M, Choi DH. Heterogeneous parallel computing for data encryption application. *6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT),* IEEE, Jeju Island, Korea, 2011; 562–566.
36. Tran NP, Lee M, Hong S, Lee SJ. Parallel execution of AES–CTR algorithm using extended block size. In *CSE*, Qu W, Lin K, Shen Y, Shi W, Hsu DF, Jin X, Lau FCM, Xu J (eds). IEEE, New York, NY, USA, 2011; 191–198.
37. Gilger J, Barnickel J, Meyer U. GPU-acceleration of block ciphers in the OpenSSL cryptographic library. In *ISC*, vol. 7483, Gollmann D, Freiling FC (eds)., LNCS. Springer: Berlin Heidelberg, 2012; 338–353.
38. Nishikawa N, Iwai K, Kurokawa T. High-performance symmetric block ciphers on CUDA. *ICNC*, IEEE Computer Society, Osaka, Japan, 2011; 221–227.
39. Iwai K, Nishikawa N, Kurokawa T. HiCrypt: C to CUDA translator for symmetric block ciphers. *ICNC,* Okinawa, Japan, 2012; 41–48.
40. Li Q, Zhong C, Zhao K, Mei X, Chu X. Implementation and analysis of AES encryption on GPU. In *HPCC-ICESS*, Min G, Hu J, Liu LC, Yang LT, Seelam S, Lefevre L (eds). IEEE Computer Society, New York, NY, USA, 2012; 843–848.
41. Niewiadomska-Szynkiewicza E, Marksa M, Janturab J, Podbielskib M. A hybrid CPU/GPU cluster for encryption and decryption of large amounts of data. *Journal of Telecommunications and Information Technology* 2012; **3**:32–43.
42. Jacquin L, Roca V, Roch JL, Mohamed A. Parallel arithmetic encryption for high-bandwidth communications on multicore/GPGPU platforms. *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, ACM, New York, NY, USA, 2010; 73–79.
43. Schönberger G, Fuß J. GPU-assisted AES encryption using GCM. In *Communications and Multimedia Security*, vol. 7025, Decker BD, Lapon J, Naessens V (eds)., LNCS. Springer: Berlin Heidelberg, 2011; 178–185.
44. Akdemir K, Dixon M, Feghali W, Fay P, Gopal V, Guilford J, Ozturc E, Worlich G, Zohar R. Breakthrough AES performance with Intel® AES new instructions, 2010. White paper, (Available from: http://software.intel.com/file/26898) [Accessed on 21 March 2014].
45. Agosta G, Barenghi A, De Santis F, Pelosi G. Record setting software implementation of DES using cuda. In *ITNG*, Latifi S (ed.). IEEE Computer Society, New York, NY, USA, 2010; 748–755.
46. Noer D, Engsig-Karup AP, Zenner E. Improved software implementation of DES using CUDA and OpenCL. *Proceedings of the Western European Workshop on Research in Cryptology (WEWORC 2011)*, Weimar, Germany, 2011; 70–76.
47. Agosta G, Barenghi A, Pelosi G. Exploiting bit-level parallelism in GPGPUs: a case study on KeeLoq exhaustive search attacks. In *ARCS Workshops*, vol. 200, Mühl G, Richling J, Herkersdorf A (eds)., LNI. GI: Bonn, Germany, 2012; 385–396.