J Cryptogr Eng
DOI 10.1007/s13389-017-0175-4

Parallel bitsliced AES through PHAST: a single-source high-performance library for multi-cores and GPUs

Biagio Peccerillo[1] · **Sandro Bartolini**[1] · **Çetin Kaya Koç**[2]

Received: 24 January 2017 / Accepted: 16 October 2017
© Springer-Verlag GmbH Germany 2017

Abstract PHAST library is a high-level heterogeneous STL-like C++ library that can be targeted on multi-core processors and Nvidia GPUs. It permits to exploit the performance of modern parallel architectures without the complexity of parallel programming. The library manages the programming and critical fine tuning of the parallel execution on both platforms without interfering with the application code structure, while maintaining the possibility to use architecture-specific features and instructions. In cryptography, performance and architectural efficiency of software implementations is crucial. This is witnessed by the extensive research in highly optimized and specialized versions of many protocols. In this paper, we assess the performance overhead and productivity improvement achievable through the PHAST library. We implement a pseudo random number generator (PRNG) based on cache-timing-attack resistant AES. We compare it with the fastest implementations in both CPU and Nvidia GPU domains. Achieved results show that the PHAST code is shorter and simpler than the state-of-the-art implementations. Its source length is 59.59% of the reference CUDA C implementation and 88.18% of the sequential C++ version for CPUs, despite being the same for both targets. It is also far less complex in terms of McCabe's and Halstead's metrics. Results show that these productivity improvements induce a limited performance overhead of the library layer: less than 5% on single-thread execution for CPUs and around 10% on Nvidia GPUs. Furthermore, performance of the PHAST PRNG automatically scales with the available cores in a nearly linear fashion, allowing programmers to fully exploit multi-core resources with the same source code.

Keywords AES PRNG · Heterogeneous programming · Multi-cores · GPUs

✉ Biagio Peccerillo
peccerillo@diism.unisi.it

Sandro Bartolini
bartolini@dii.unisi.it

Çetin Kaya Koç
koc@cs.ucsb.edu

[1] Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche, Università degli Studi di Siena, Siena, Italy

[2] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

1 Introduction

Replication of components is nowadays the main driver of hardware architectural improvements. Parallel devices dominate the market and deliver ever-increasing raw computational capabilities [14,31]. However, the theoretical aggregate performance of such devices can be exploited only through fairly complex parallel programming strategies, which typically need to be fine-tuned on the specific architecture to fully harness its potential.

The most common classes of parallel systems are multi-core processors and GPUs, that can often be found in the same system, making it a so-called *heterogeneous* system. Since GPUs are no longer limited to graphics-related computations, both multi-cores and GPUs can be programmed to solve general purpose problems. This gives programmers an additional degree of freedom (and complexity), allowing them to take advantage of few complex units (multi-cores) and/or many simple units (GPUs) in parallel, depending on the *application problem* at hand (e.g., cryptography, physics, computer vision).

However, there is not a single native approach that allows programmers to write code compatible with both multi-

Published online: 29 October 2017

cores and GPUs, or simply with GPUs of different vendors. Coding style, approach and optimization are very specific of the target, almost never reusable on another one and absolutely mandatory for performance. For instance, while multi-cores can be programmed in terms of standard C++11 threads [15], the native language of Nvidia GPUs is CUDA C [30], a subset of C++ with proprietary extensions shipped with its own compiler that requires programmers to express their program in terms of kernels, blocks of threads, etc. Heterogeneous, non-native solutions exist, but they usually fail in the purpose of providing a simple, familiar interface that permits the production of efficient code with limited effort. It is the case, for instance, of OpenCL [18], which is considered the de-facto standard of heterogeneous programming solutions, but being quite low-level for performance reasons, needs nontrivial specific expertise to be mastered. So it cannot be considered a widely adoptable technology. We define a technology as *widely adoptable* if its usability features and abstractions make it effectively mastered by experts of sequential programming in a specific application domain (e.g., cryptography, image processing, finite element simulation, finance), without requiring them to be experts of parallel and heterogeneous programming.

In this paper we present Parallel Heterogeneous Architecture STL-like Template (PHAST) library, which is a high-level C++ library which allows to write efficient and concise code *once*, through high-abstraction mechanisms and in a sequential fashion, letting the library manage the targeting toward heterogeneous parallel platforms, i.e., currently multi-core CPUs and Nvidia GPUs. Furthermore, architecture-specific parallelization and tuning parameters can be adjusted in a way that is independent from application code.

To assess the potential of PHAST library, we consider the implementation of a cache-timing-attack resistant AES-based pseudo random number generator [16,21] as it is a challenging real-world application with many solutions available in the literature, which are highly specialized for specific architectures.

The major contribution of this paper can be summarized as in the following:

- analysis of the efficiency and expressiveness of PHAST code by comparing it against state-of-the-art implementations of AES-based PRNGs. Results show that the *general* code is able to run within about 10% from the highly optimized state-of-the-art versions on various platforms;
- exemplification of the orthogonality in PHAST between application code and parallelization/tuning parameters. We show both performance scaling with the number of cores in multi-cores, as well as the possibility to identify the optimal parameter set, and board resource usage strategies, for different Nvidia boards;
- evaluation of the productivity improvement achievable using the PHAST library through the measurement of well-known code productivity metrics [9,23] in comparison with the considered state-of-the-art implementations. The achieved improvements (e.g., 8 vs 11 and 68 cyclomatic complexity for multi-cores and Nvidia GPUs, respectively) are significant in themselves but appear even more interesting considering the small induced overhead and the single-source code.

The rest of the paper is organized as follows. In Sect. 2 we summarize the story and evolution of parallel architectures to the present day. In Sect. 3 we present pseudo random number generation based on AES and the implementations that achieve best performance. In Sect. 4 we present PHAST library and our implementation of an AES-based PRNG. In Sect. 5 we compare the performance of our implementation against the best implementations on multi-cores and Nvidia GPUs and we also show code productivity metrics of all the involved PRNGs. Eventually, in Sect. 6 we present the conclusions of this paper.

2 Current parallel architectures

From the Eighties until a decade ago the constant improvement in uniprocessors was the main reason for performance improvement in computing architectures. The main technological advancements that drove this process were diverse, with frequency scaling, execution optimization, and cache memories being the principal ones [37]. Two out of three seemed to have come to an end around 2004:

- Frequency scaling was considered over when heat dissipation, power consumption, and leakage power reached a critical threshold [37];
- Execution optimization—i.e., doing more work per cycle—had been deeply exploited and explored till nowadays from various perspectives even if improvements are typically not as big as in the past [37]. Further advancements in ILP through complex execution engines, in particular, diminished their attractiveness mainly due to power growing faster than performance [12].

At a certain moment in time, improvements in cache technology were the only ones likely to go on [37], to capture bigger and bigger working sets and thus hide main memory latency. However, with Moore's Law alive, processor manufacturers had to find a way to employ the enormous, ever-increasing number of transistors [6,37].

Also due to the emerging wire-delay issues [19], which prevented big structures on chip to be kept synchronized,

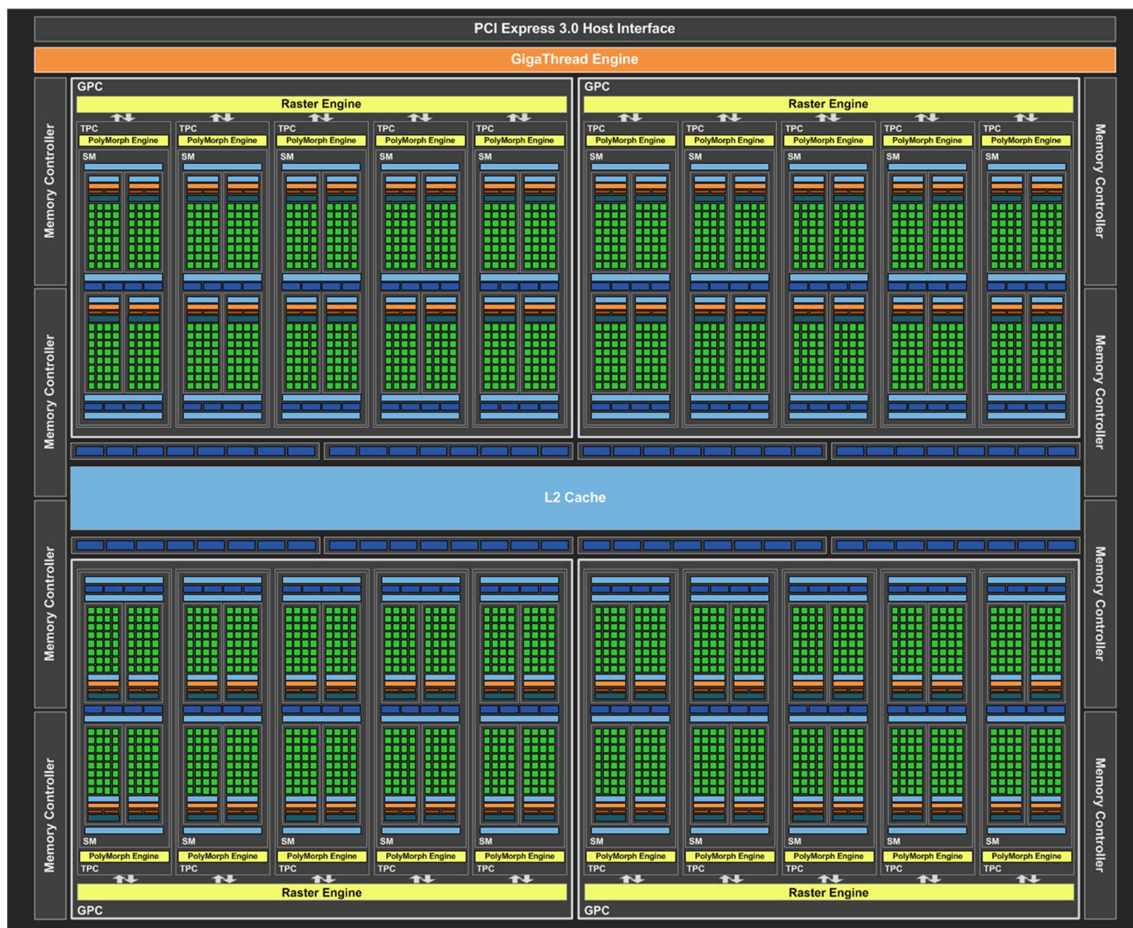

Fig. 1 Block diagram of the GP104 Nvidia GPU core. Green squares are CUDA cores organized into 20 Streaming Multiprocessors (SM). Each SM combines 128 CUDA cores, 256 KB register file, 96 KB of shared memory, 48KB of L1/texture cache and eight texture units (color figure online)

the traditional processor design paradigm faded and the new trend in performance scaling turned out to be the replication of independent cores on the same chip, thus investing in *parallel* architectures [6]. This design technique gives the possibility to harness the power of a growing number of transistors, also increasing the number of operations per unit time without increasing power demand. Two real-world architectures exemplifying this trend are multi-cores and Graphics Processing Units (GPUs), which push this trend much further than CPUs due to the finer grain at which replication of computing structures is performed.

2.1 GPUs

GPU market is dominated by AMD and Nvidia graphic cards. For the purpose of this paper, we will focus on Nvidia cards. The support of AMD cards is planned as future works and it is expected to take advantage from the architectural similarities among the two.

Modern GPUs are based on an array of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs) and memory controllers [31], as it is exemplified in Fig. 1 for GP104 series, which is adopted within GeForce GTX 1080 GPU. Every GPC has a Raster Engine and some SMs and can be regarded as a self-contained GPU [28]. SMs are highly parallel multi-core processing units, with tens to hundreds of simple CUDA cores, thousands of registers, a unified shared memory unit, a unified L1 cache memory, some texture units, some Special Function Units (SFUs), some Load/Store units and some warp schedulers that select fixed-size groups of parallel threads (called “warps” in Nvidia terminology) and issue instructions to CUDA cores[1] [28,31]. CUDA cores are relatively simple scalar processors with a pipelined ALU and a pipelined FPU which support common 32-bit operations, single- and double-precision floating point data [28]. Memory controllers are tied to some Raster Operation (ROP) units and an L2 unified cache that services all load, store and texture requests [28]. They are connected to the memory die that features a fast GDDR memory, GDDR5X in its most recent incarnations [31].

[1] The exact amount of these components depends on the generation and the model of the graphic card.

The execution model is a classical Single Program Multiple Data (SPMD), the most common style of parallel programming [25]. SMs are responsible for executing blocks of threads by scheduling warps to CUDA cores [31]. *Warps are the atomic units of execution*, so that the individual threads in a warp execute in a SIMD fashion [10,25]. Branch divergences inside a warp are serialized and synchronized when all paths are executed, with NOPs being forwarded to inactive threads [10].

Nvidia GPUs' programming language is CUDA C, a C++ subset (no virtual functions, no function pointers, no standard library, etc) with some extensions, mainly storage and function annotations and a special syntax to offload "kernel functions" on the GPU—which is treated as a co-processor [30]. CUDA programs are divided in two separate sections, depending on where their execution takes place: host code and device code. In host code programmers allocate memory, transfer data to/from the GPU and launch kernel functions to be executed on the GPU. Device code is mainly kernel functions (invoked from host code, execute on the device), device functions (invoked from kernel or device functions, execute on the device) and global data conveniently annotated. When launching a kernel, programmers must specify the size of the grid of execution in terms of number of blocks, the size of each block, the amount of dynamic shared memory to use and the particular stream of execution [30]. Inside a kernel, threads manipulate global[2] data, usually depending on their indexes, and can be synchronized block-wise via an API call. A fast shared memory can be used as a scratchpad to accelerate accesses or to communicate block-wise [30].

Apart from the aspects related to CUDA programming model, there are other aspects programmers need to care about to write optimum code from the performance viewpoint. Block sizes must be carefully chosen, global memory must be accessed in an aligned way [29], shared memory has a relatively small size and must be wisely managed, and accessed, so to avoid memory bank conflicts [29], the convenience of using constant or texture memory must be taken in account [29]. Therefore, these many degrees of freedom make writing highly effective CUDA code challenging.

2.2 Multi-core processors

Multi-cores are the natural evolution of uniprocessors. The term can be referred to single-chip systems with multiple cores—often called "multi-cores"—or to multiple-chip computers, each of which can be a multi-core [12]. The number of cores can vary from two to dozens, and the communication between them is essentially achieved via one or more levels of shared cache/memory [12]. Especially in high-end ones, each core is usually an autonomous super-scalar processor with a long pipeline, out-of-order execution, branch-prediction, vector registers and units, and two levels of cache, with the third level shared between the cores—it is, for instance, the case of the Intel i7 series processors [14].

From a Flynn's taxonomy point of view, multi-cores can be thought as shared memory MIMD machines [25]. They are capable of executing independent programs or independent flows of execution of the same program at the same time.

In the days of vertical scaling, programs used to gain performance when a new processor hit the market just because of the greater clock frequency. To achieve a similar effect nowadays, programmers need to take explicit advantage of the increased number of parallel cores within the chip. Specifically, programs need to be written and organized into multiple concurrent flows of execution. The code must be structured in a flexible number of threads of executions, so to guarantee load balancing and maximum utilization of the underlying hardware as it evolves (e.g., increasing the number of cores). However, as a matter of fact, concurrent programs are hard to write and very hard to debug [37].

C++ natively supports parallel programming since C++11 [15]. Before the standardization of threads, locks, mutexes, etc. programmers were forced to use non-standard, often platform-dependent libraries and frameworks such as PThreads [27], Win32 Threads [24], OpenMP [3], Boost. Threads [35], Intel TBB [34]. Nevertheless, despite standardization guarantees code-portability, it does not make concurrent programming any simpler. Moreover, C++11 does not provide any APIs to directly manage thread affinity [15], an aspect that can have a significant impact on performance as we will briefly show in Sect. 5.

2.3 Heterogeneous programming approaches

A system consisting in different kinds of computing devices is said to be a *heterogeneous* system. It is the case, for instance, of most today's PCs equipped with a multi-core processor and a GPU. A programmer willing to run an application on both of these devices to achieve maximum performance, would face the necessity to code it in a manner that is compatible with both of them. As suggested in this section, adopting standard approaches would mean writing the application more than once. Doing so would violate the *Don't Repeat Yourself* (DRY) principle [13], would increase the amount of code to maintain, would be error-prone and, most importantly, time-consuming. To solve this problem in a more productive way, a number of heterogeneous libraries and frameworks have been proposed so far. A quick digest of them follows.

OpenCL [18] is a C99 CUDA-like approach to heterogeneous programming. It is considered the de-facto standard, but its low-level nature makes its wide adoption difficult.

[2] Within the global memory of the video card.

Some other approaches prefer to delegate code parallelization to the compiler to limit the amount of setup code, but they also require the user to abandon his/her compiler of choice. It is the case of C++ AMP [7], an open extension to C++11 provided with a Microsoft implementation. It defines an `array_view` object on which the programmer can invoke a `parallel_for_each` algorithm. Another example is PACXX [8], a unified programming model implemented as a custom C++14 Clang-based compiler. Parallel computation can be expressed via `std::async` on vectors or via function annotations. OpenACC [32] is an OpenMP-like framework that requires the code to be annotated with special pragmas, thus defining its own language and constructs to be interleaved with application code. Other approaches prefer to wrap a low-level solution with high-level constructs. EasyCL [33] is a thin OpenCL wrapper that defines some macros and classes to manage part of the setup code under-the-hood. SYCL [17] is a C++14 layer on top of OpenCL that supports kernel functions as lambdas. It is a high-level library, but it still needs the programmer to express him/herself via OpenCL terminology. SkePU [5] and SkelCL [36] are C++ template libraries based on common algorithmic patterns called *skeletons*. The former has been implemented on top of CUDA, OpenCL, and OpenMP, while the latter on top of OpenCL only. They both need the user to write application code in terms of skeletons. ArrayFire [38] is an array-centric high-level wrapper of CUDA or OpenCL. It defines an `array` class and it permits to express computation via a math-resembling syntax. Boost.Compute [22] is an STL-like library built around OpenCL. It offers containers, algorithms and iterators and it supports lambda functions as user-defined kernels. Kokkos [4] is a high-level C++11 library that wraps CUDA, Pthreads or OpenMP. It defines multi-dimensional arrays accessible via views. Programmers can express their computations by passing functors or lambdas to `parallel_for` or `parallel_reduce` algorithms.

In this paper we propose PHAST, a high-level heterogeneous C++ library. It is implemented on top of CUDA, Boost.Threads, or C++11 threads and can be targeted on Nvidia GPUs or multi-cores. It is based on mono-, bi- and three-dimensional containers that can be accessed via different kinds of iterators. Computation is achieved by calling many STL-resembling algorithms (e.g., `for_each`, `transform`, `sort`, `find`, …) on ranges of iterators, also supporting user-defined functors where needed. We also give the possibility of calling in-functor algorithms on portions of the iterated containers so to allow a finer grain of parallelism. Parallelization parameters can be adjusted at need by calling API functions in a way that is independent from application code. Moreover, the use of PHAST library does not prevent the use of architecture-specific constructs that could help performance.

To prove the value of PHAST, in this paper we are addressing a *challenging* AES-based PRNG. It is a complex application that required the work of many researchers to be optimized to the level we have at the present day in highly tuned low-level implementations, as we will show in the next section.

3 AES and its implementations

Pseudo random number generators (PRNGs) are algorithmic ways to generate sequences of numbers that can be reasonably considered random in specific application contexts. As Donald Knuth observes, these sequences are important in many kinds of applications, such as simulation, sampling, numerical analysis, computer programming, decision making, cryptography, aesthetics, and recreation [20]. The Advanced Encryption Standard (AES) is a cryptographic algorithm that has been standardized in 2001 [26]. It has many properties such as speed, nonlinearity, and portability that make it a good fit for a high-quality PRNG [11].

Many AES implementations have been proposed so far. They are equivalent from a statistical point of view, but can differ in quality by other means. A common requirement of a *good* AES implementation is the immunity from side-channel attacks. Since many implementations make use of lookup-tables for performance reasons, this is not a common feature. However, recent implementations proved immune to cache-timing attacks by replacing lookup-tables and data-dependent branches with equivalent operations in Galois fields [16]. For instance, the so-called *SubBytes* step is an inversion in GF(256) [16].

Rewriting the entire algorithm as a series of Galois field operations means implementing AES as a sequence of atomic boolean instructions [2]. In this case, the most natural representation of AES bytes would be as 8 boolean variables each. Obviously, the performance loss would make the whole procedure impractical. A solution comes from a technique known as *bit-slicing* [16,21]. It is achieved by rearranging the bits of many AES blocks in groups, so that equivalent bits of multiple bytes are packed in the same register. This way, the single atomic boolean operations can be performed between registers of a convenient size depending on the underlying architecture. For instance, in [16] Käsper et al. describe a fast AES implementation that takes advantage of 128-bit wide XMM registers and SSE instructions. Bit-slicing is, in the end, a technique to achieve bit-level parallelism and to process more AES blocks at once.

Another source of parallelism in AES can come from its modes of operation. Since it is a block cipher, it has many modes of operation that regulate how multiple blocks are encrypted. The most appropriate mode for pseudo random number generation is CTR [11]. ECB mode is similar to CTR

and it also can be used to implement a PRNG [21]. Both CTR and ECB modes require AES blocks to be encrypted independently, so both of them expose an intrinsic parallelism. Programmers can take advantage of it, for instance by using the multiple cores of a chip-multiprocessor or of a GPU.

3.1 State-of-the-art implementations

We refer to [16] and to [21] as state-of-the-art implementations of AES, respectively, multi-core-compatible and GPU-compatible.

Käsper et al. [16] implement a cache-timing-attack resistant bitsliced AES encryption in counter mode for 64-bit Intel processors. Their implementation is 25% faster than the best of the previous ones, with 7.59 cycles per byte on an Intel Core 2. Their implementations can be found on authors' websites, written in assembly and QHASM.

Lim et al. [21] describe a deterministic AES-based PRNG written in CUDA C. Their implementation is a cache-timing-attack resistant bitsliced AES in ECB mode. It achieves 78.6 Gbps on a GeForce GTX 480, 31-62% faster than the fastest previous implementation on similar devices. In their code, the authors addressed many common problems like coalesced accesses to main memory, shared memory conflict avoidance, fine register allocation to avoid spilling and maximize occupancy, and so on.

Both implementations are low-level, deeply optimized in the respective domains, and achieve great performance. We implemented a general software implementation of a PRNG based on AES in ECB mode through PHAST, our high-level heterogeneous library. The same source code will be used for both Nvidia GPUs and multi-cores, and we will analyze the performance differences with the above mentioned highly specialized versions. In the next section we present our library and our PRNG implementation, then we show its performance on different architectures. Eventually, we will also compare programming effort metrics on the considered implementations to highlight which programming approach can be more effective from the productivity point of view.

4 PHAST library

PHAST library is a high-level heterogeneous C++ library. Its inner layers are implemented in CUDA C, Boost.Threads or C++11 threads and currently allows targeting on Nvidia GPUs or multi-cores. These layers are not part of the interface, so users can express their code in a platform-independent way. In fact, PHAST programmers can code their applications in terms of containers, iterators, and algorithms in an STL-like fashion, thus using common sequential techniques.

We provide three kinds of containers: mono-dimensional `vector`, bi-dimensional `matrix`, and three-dimensional `cube`. Each of them can be accessed via different kinds of iterators. For instance, a `cube` container having dimensions $M \times N \times O$ can be iterated in the following ways:

- $M \times N \times O$ scalar iterators—each one pointing to one element;
- $M \times N$ vector iterators—each one pointing to an O-dimensional vector;
- M matrix iterators—each one pointing to an $N \times O$-dimensional matrix;
- grid iterators—each one pointing to a sub-cube in a tiling fashion.

Ranges of iterators can be used in the many parallel algorithms PHAST library provides, e.g., `replace`, `copy`, `generate`, `find`, ... Some algorithms accept user-defined functors, in which programmers can personalize their computation at need. Library macros permit the definition of different kinds of functors, each kind being specialized on a different *slicing method* of the container. So, for instance, a vector-functor will work in an algorithm paired with ranges of vector iterators, and will apply computation to each of the vectors identified by the range of iterators *in parallel*. Inside the body of a functor, users can take advantage of *in-thread* versions of the aforementioned STL-resembling algorithms, so keeping their code concise even in functors.

Different platforms lead to different parallelizations, regulated by different platform-specific parameters. PHAST tries to infer these parameters, but it does not prevent the users from explicitly specifying them. So, various API calls can be invoked by the programmers to allow fine tuning. These API calls are independent from application code, can have an impact on performance but do not interfere with the correctness of the program.

Users could be willing to specialize some portions of their code according to the underlying device. PHAST does not prevent them from doing so, and low-level architecture-specific optimizations are still possible under the scope of `ifdefs` or similar constructs.

In brief, the purpose of PHAST library is:

- to allow programmers to write parallel heterogeneous code at a high-level of abstraction, not worrying about:
  - the degree and nature of parallelism of the generated and executed code (e.g., number of threads and their relationship with the data they operate on);
  - the architectural and language peculiarities of the target devices (Nvidia GPUs or multi-cores).
- to reach near-native performance on the target devices, possibly with some application code-independent fine parameter tuning;

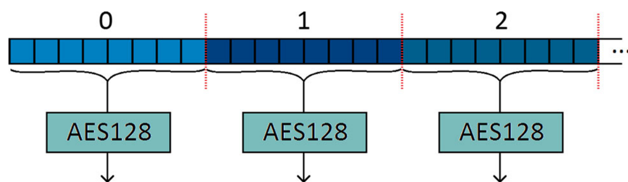

Fig. 2 A grid of eight elements is applied to vector data, so to iterate over chunks of eight elements

– to not shield application code from low-level optimizations such as:

  - potential architecture-specific algorithmic refinements that can be found in the literature, e.g., code replication on GPUs, branching on multi-cores;
  - target-specific ad-hoc instructions, e.g., `__byte_perm` in CUDA code, `_mm_shuffle_epi8` in SSE-compatible-architecture code.

4.1 Bitsliced implementation of AES using PHAST library

In AES block-cypher, data encryption is achieved by applying four basic operations ten to fourteen times, depending on the desired cryptographic strength required, on a data block. Such operations are commonly known as SubBytes, MixColumns, AddRoundKey, and ShiftRows [26]. We have implemented a general AES code version using PHAST, in which we have applied specific optimizations, inspired by the highly tuned versions described in [16,21] for multi-cores and GPUs, respectively.

AES core is a `for_each` algorithm that works in parallel with chunks of eight variables at once as seen in the cited papers. To achieve this, a grid of eight elements is applied to the data-vector and then iterated as shown in Fig. 2 and in the following excerpt.

```
// declare data−vector
phast::vector<uintN_t> data(n);
// define a grid of sub−vectors of size 8
phast::grid<phast::vector<uintN_t> > data_grid(data, 8);
// aes128_func is applied to each sub−vector in parallel
phast::for_each(data_grid.begin(), data_grid.end(), aes128_func);
```

`aes128_func` is an instance of the user-defined functor `aes128`. We are considering here the 128-bit version of AES, thereby comprising 10 rounds of the basic operations. The functor is declared in a separate header file by using PHAST macros as shown below.

```
// main functor
_FUNCTOR_HEAD(aes128)
    // in−thread container − can be accessed in device code
    phast::matrix_th<uintN_t> round_keys;
_VEC_BODY(data)
    uintN_t x0, x1, x2, x3, x4, x5, x6, x7;
    // data from phast::vector_th data are bitsliced and
    // saved in x0...x7 variables
    bitslice(data, x0, x1, x2, x3, x4, x5, x6, x7);

    add_round_key(x0, x1, x2, x3, x4, x5, x6, x7, round_keys, 0);
    // round 1−9
    for(int round = 1; round <= 9; ++round)
    {
        sub_bytes(x0, x1, x2, x3, x4, x5, x6, x7);
        shift_rows(x0, x1, x2, x3, x4, x5, x6, x7);
        mix_columns(x0, x1, x2, x3, x4, x5, x6, x7);
        add_round_key(x0, x1, x2, x3, x4, x5, x6, x7,
            round_keys, round);
    }
    // round 10
    sub_bytes(x0, x1, x2, x3, x4, x5, x6, x7);
    shift_rows(x0, x1, x2, x3, x4, x5, x6, x7);
    add_round_key(x0, x1, x2, x3, x4, x5, x6, x7, round_keys, 10);

    // x0...x7 values are retrieved and stored back in
    // phast::vector_th data
    ibitslice(data, x0, x1, x2, x3, x4, x5, x6, x7);
_FUNCTOR_TAIL
```

In functor code the four AES phases are clearly visible. As an example, we also show the implementation of the ShiftRows phase.

```
_PHAST_FUNCTION void shift_row(uint2_t& data)
{
    uint2_t tmp(data);
    data.at(0) = (tmp.at(0) & 0x0000ffffu)
        | ( (tmp.at(0) >> 4) & 0x0fff0000u)
        | ( (tmp.at(0) << 12) & 0xf0000000u);
    data.at(1) = ((tmp.at(1) << 8) & 0x0000ff00u)
        | ( (tmp.at(1) >> 8) & 0x000000ffu)
        | ( (tmp.at(1) >> 12) & 0x000f0000u)
        | ( (tmp.at(1) << 4) & 0xfff00000u);
}

_PHAST_FUNCTION void shift_row(uint4_t& data)
{
    uint4_t tmp(data);
    // data.at(0) = tmp.at(0);
    data.at(1) = (tmp.at(1) >> 8) | (tmp.at(1) << 24);
    data.at(2) = (tmp.at(2) >> 16) | (tmp.at(2) << 16);
    data.at(3) = (tmp.at(3) >> 24) | (tmp.at(3) << 8);
}

_PHAST_FUNCTION void shift_rows(uintN_t& x0, uintN_t& x1,
    uintN_t& x2, uintN_t& x3, uintN_t& x4,
    uintN_t& x5, uintN_t& x6, uintN_t& x7)
{
    shift_row(x0);
    shift_row(x1);
    shift_row(x2);
    shift_row(x3);
    shift_row(x4);
    shift_row(x5);
    shift_row(x6);
    shift_row(x7);
}
```

Function overloading permits the definition of two `shift_row` functions, the right one being called depending on the resolution of the typedef `uintN_t`, which expands differently on different platforms: `uint2_t` on Nvidia

Table 1 Different operator expansions on different devices

	Multi-cores	Nvidia GPU
a + = b;	a.data = _mm_add_epi32(a.data, b.data);	a.data[0] + = b.data[0];
		a.data[1] + = b.data[1];
		a.data[2] + = b.data[2];
		a.data[3] + = b.data[3];

GPUs, a vectorial type wrapping two unsigned integers, and `uint4_t` on multi-cores, a vectorial type wrapping four unsigned integers, as in the following.

```
#ifdef _PHAST_USING_CUDA
typedef uint2_t uintN_t;
#else
typedef uint4_t uintN_t;
#endif
```

Introducing a typedef that expands differently on different architectures was not a necessary choice in terms of code correctness, since both `uint4_t` and `uint2_t` are fully supported on multi-core processors and Nvidia GPUs. All the application code could be implemented in terms of any of them without hampering heterogeneity, but performance would have suffered by this choice. In fact, looking at the code of the two state-of-the-art implementations, different kind of variables are used: 128-bit integer SSE vector (i.e., `__m128i`) in Käsper et al. implementation, that naturally maps on `uint4_t`, and `vec2` in Lim et al. implementation, a class wrapping a CUDA `uint2` vectorial type with constructors and operator overloads that naturally maps on `uint2_t`. These types defined in PHAST library closely recall CUDA vectorial types, but are richer than them for two reasons:

- PHAST types are mapped on SSE types if available, general purpose registers otherwise;
- PHAST types are shipped with arithmetic operator overloads and constructors.

So, for instance, a `uint4_t` variable wraps an `__m128i` on SSE-capable architectures. In Table 1 an example of different operator expansions according to the underlying device is given.

ShiftRows is a straightforward function that didn't require much effort to be implemented. AddRoundKey is also quite simple, requiring only eight XOR operations between the bitsliced variables and the round keys. These operations were expressed using classical C++ syntax, taking advantage of the operation overloading on PHAST vectorial types. MixColumns bitsliced implementation can also be expressed via common XOR operations as can be seen in appendix A in [16]. SubBytes, being the only phase that is not linear in GF(256), is maybe the most studied AES phase and many implementations can be found in the literature [1,2,16,21]. In particular, we implemented a SubBytes based on Boyar et al. work, clearly described in [1], and one based on Lim et al. work, thanks to the source code the authors made available for our work. Both of these implementations, as well as the other AES phases, are platform-agnostic and can be run on GPUs and multi-cores. For the performance and productivity analysis, we adopted the second one.

ShiftRows phase is a fitting example of the possibility to optimize PHAST code by using low-level architecture-specific constructs. In fact, this operation can take advantage of low-level instructions on SSSE3-compatible processors (`_mm_shuffle_epi8`) and on CUDA devices (`__byte_perm`). In the following, *ISA versions* is used to denote the implementations that take advantage of architecture-specific optimizations, while *non-ISA versions* or *plain versions* are used to denote the others.

In the following code excerpt, both `shift_row` functions (one for each managed vectorial type) have been optimized to take advantage of architecture-specific instruction sets. For each of them, the optimizations have been put under the scope of the *if* branch of an `ifdef` clause that checks what architecture is being used. Conversely, the non-architecture-specific version has been maintained under the scope of the *else* branch. This way, both these functions are still multi-platform in their nature and can be executed on multi-cores and Nvidia GPUs, but they also give the possibility to execute optimized code when executed on a specific architecture.

```
_PHAST_FUNCTION void shift_row(uint2_t& data)
{
#if defined(_PHAST_USING_CUDA)
    data.at(0) = __byte_perm(data.at(0), data.at(0) >> 4, 0x7610u)
        | ((data.at(0) << 12) & 0xf0000000u);
    data.at(1) = __byte_perm(data.at(1), data.at(1) >> 4, 0x6701u)
        | ((data.at(1) << 4) & 0x00f00000u);
#else
    uint2_t tmp(data);
    data.at(0) = (tmp.at(0) & 0x0000ffffu)
        | ((tmp.at(0) >> 4) & 0x0fff0000u)
        | ((tmp.at(0) << 12) & 0xf0000000u);
    data.at(1) = ((tmp.at(1) << 8) & 0x0000ff00u)
        | ((tmp.at(1) >> 8) & 0x000000ffu)
        | ((tmp.at(1) >> 12) & 0x000f0000u)
        | ((tmp.at(1) << 4) & 0xfff00000u);
#endif
}

_PHAST_FUNCTION void shift_row(uint4_t& data)
{
#if defined(_PHAST_USING_MULTI_CORE) && defined
    (_PHAST_SSSE3)
    static const uint4_t mask(0x03020100u, 0x04070605u,
        0x09080b0au, 0x0e0d0c0fu);
    data = _mm_shuffle_epi8(data.data, mask.data);
#else
    uint4_t tmp(data);
    // data.at(0) = tmp.at(0);
```

```
    data.at(1) = (tmp.at(1) >> 8) | (tmp.at(1) << 24);
    data.at(2) = (tmp.at(2) >> 16) | (tmp.at(2) << 16);
    data.at(3) = (tmp.at(3) >> 24) | (tmp.at(3) << 8);
#endif
}
```

This section showed how PHAST library can improve programmers' productivity in writing high-level cross-platform parallel code. It analyzed the case of an AES-based PRNG showing how users can focus on application code *once* and obtain an implementation that can run on Nvidia GPUs and multi-cores. The next section shows a performance comparison between PHAST implementations and state-of-the-art implementations discussed in Sect. 3.1.

5 Results

Since PHAST code can run on multi-cores and Nvidia GPUs, we tested our implementation on both classes of devices.

5.1 Multi-cores

The multi-core-based machines employed for the experiments are summarized in Table 2.

Our PHAST implementation, which can run on both CPUs and Nvidia GPUs, is inspired by Käsper et al. [16] for MixColumn step, which is described in the appendix of their paper [16]. While for SubBytes we implemented the pseudocode described in [1] because it is one of the fastest and general. Käsper et al. is probably the fastest but it is not general as it explicitly requires SSSE3 ISA support.

Table 2 Multi-core-based machines used for benchmarking

	meeseeks	maxi
CPU	Intel Core i7-4790K	Intel Xeon E5-2650 v2
CPU type	quad-core hyper-threaded	dual octa-core hyper-threaded
CPU frequency	4.00 GHz	2.60 GHz
RAM	16.0 GB	64.0 GB
OS	14.04.1-Ubuntu 3.16.0 x86_64	Debian 3.16.7 x86_64
	elwood	**golia**
CPU	AMD Phenom II X6 1100T	Intel Core i7-6800K
CPU type	hexa-core	hexa-core hyper-threaded
CPU frequency	3.30 GHz	3.60 GHz
RAM	16.0 GB	128.0 GB
OS	Debian 3.2.65 x86_64	Debian 3.16.0 x86_64

Table 3 Performance comparison (Gbps) between mono-thread AES PRNGs on CPUs

	no-lib plain	PHAST plain	no-lib ISA	PHAST ISA	Käsper–Schwabe
meeseeks	1.85	1.82	3.30	3.26	5.09
maxi	1.37	1.31	2.42	2.33	3.99
elwood	0.56	0.59	–	–	–
golia	1.53	1.46	2.65	2.56	4.18

Käsper–Schwabe results employ a different algorithm for SubBytes AES step

For multi-cores, as an overall reference, we report also the highly tuned and architecture-specific, bitsliced AES encryption in counter mode described in [16] (Käsper–Schwabe in the following) as it constitutes a landmark in terms of absolute performance for CPUs, despite not being usable on different architectures (e.g., multi-cores without the required x86 SSE ISA support, and GPUs). Its source code available at the authors' site is a thin C function wrapper that invokes assembly code. It is a thread-safe function, but the whole benchmark is a sequential implementation and no work-partition or thread management is given. For this reason, Table 3 shows a performance comparison on different multi-cores with PHAST implementations launched with thread-number parameter set to 1.

Then, PHAST code has been implemented in two slightly different versions: one that takes advantage of low-level architecture-specific SSE instructions (ISA version) and one that does not (plain version), to witness the library capability to reach the machine-specific features when needed.

Furthermore, we implemented the reference code using the exact same versions of all the algorithms employed in the PHAST implementation, but without the support of PHAST library. Again with two minor variations adopting SSE extensions (no-lib ISA) or not (no-lib plain). This way it is possible to assess the exact overhead induced by the abstractions and heterogeneous parallelization facilities provided by PHAST library.

We underline that, since elwood is not an SSSE3-compatible processor, the only version that can run on it is the plain PHAST version. In fact, it does not use architecture-specific instructions and it is guaranteed to be portable across various multi-cores.

Table 3 shows that in single-thread conditions on the same code, the maximum library overhead amounts to 4.79% on golia and that can be regarded as a measure of the performance impact of PHAST library on single-core applications. Both ISA-enhanced and plain versions highlight a similar limited overhead across different CPUs. The Table highlights also a quite big difference in performance between Käsper–Schwabe and PHAST, with Käsper–Schwabe 1.7 times faster than PHAST ISA in the worst case. This performance differ-

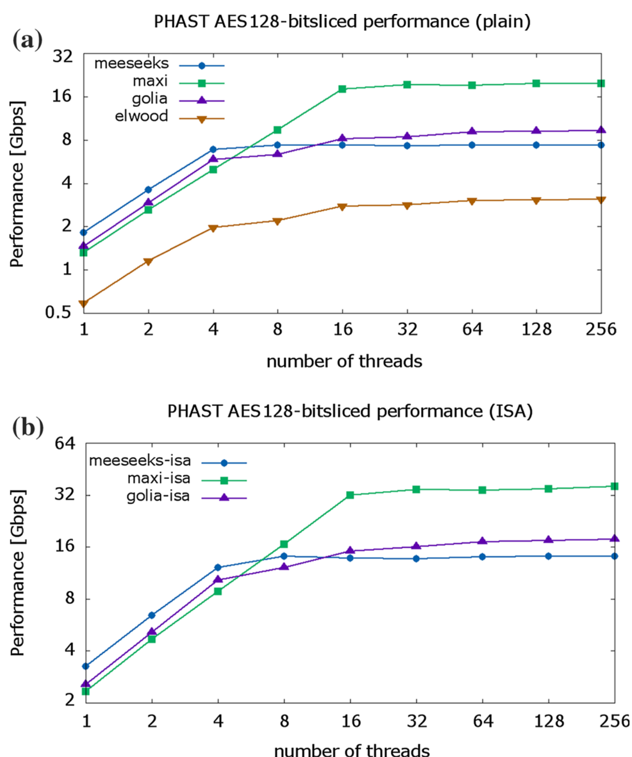

Fig. 3 AES performance on multi-cores using PHAST library in correspondence of different numbers of threads. **a** Plain versions. **b** ISA versions

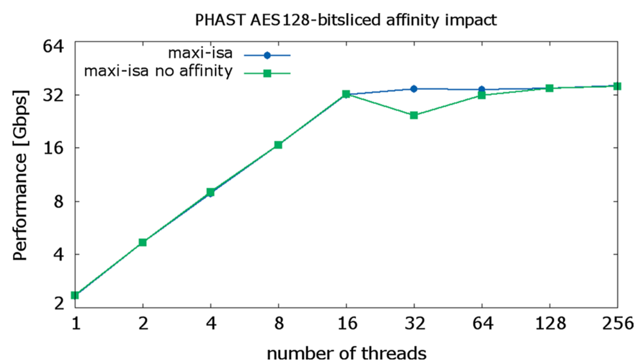

Fig. 4 The impact of affinity management on multi-core performance

Table 4 Nvidia GPUs used for benchmarking

	gtx970	gtx1080
GPU	GeForce GTX 970	GeForce GTX 1080
Compute Capability	5.2	6.1
Global Memory	4095 MB	8113 MB
GPU frequency	1.25 GHz	1.85 GHz
Memory frequency	3505 MHz	5005 MHz
Number of CUDA cores	1664	2560
Host	meeseeks	golia

ence is mainly due to the SubBytes algorithm which in case of Käsper–Schwabe is extremely tuned in assembly for full exploitation of the specific SSSE3 ISA support available in some CPUs.

Furthermore, one of the main advantages of the proposed PHAST library is the ability to automatically exploit the aggregate computational power of multi-cores using more than a single thread. This approach requires that the underlying hardware has multiple processing units (e.g., cores in CPUs) and that all of them have *enough* work to do for hiding the costs of thread allocation and management. PHAST library can determine such hardware parallelism automatically or the user can explicitly call an API function to force the desired number of runtime threads. Figure 3 shows the performance variation when varying the number of threads used. Each of the multi-cores used achieves an almost linear performance scaling up to the number of available *physical* cores, with a further smaller gain in processors having hyper-threading technology (all but elwood machine with AMD Phenom II processor). From the methodological viewpoint, the amount of data each thread works with has been kept constant so to avoid 'noise' due to caching effects in the cores.

The intrinsic support of parallel execution with limited overhead, allows PHAST implementation to automatically reach far better performance than the Käsper–Schwabe single-thread version and, most of all, allows to trivially harness the increasing overall computational power available in successive generations of processors.

As previously cited, PHAST library also manages thread affinity. It is another user-adjustable parameter, but default behavior is usually the best choice. It is set to spread the available threads across the physical packages, then across the physical cores and uses hyper-threading only as a last resource. This way thread locality is minimized and the maximum amount of cache space and functional units is assigned to every thread. The impact on performance of affinity can be seen in Fig. 4, where the green curve has been obtained with affinity management shut down. It can be seen that for AES, despite affinity management impact on performance is negligible when few threads are used, it becomes critical when all the logical cores are occupied.

5.2 GPUs

The GPUs used are summarized in Table 4. In the GPU case, PHAST provides many user-adjustable parameters that reflect the degrees of freedom CUDA model defines. All of these parameters can be varied without altering the structure of the source code. The considered parameters are:

– Major Blocksize: used to set the leading dimension of CUDA blocks of threads;

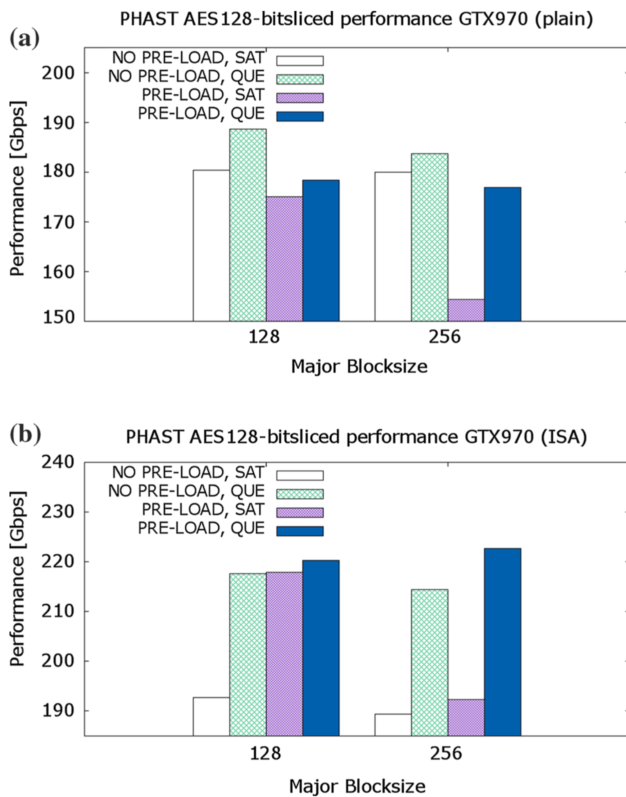

Fig. 5 AES performance on GeForce GPU GTX970 in correspondence of different combinations of Major Blocksize, Shared Pre-Load and Scheduling Strategy. **a** GTX970 plain version. **b** GTX970 ISA version

- Minor Blocksize: used to achieve a finer grain of parallelism in the case of in-functor algorithm invocation;
- Scheduling Strategy: can be set to achieve the maximum occupancy of blocks that loop over the workload (`SATURATE`) or to allocate as many blocks are needed and to delegate their scheduling to hardware scheduler (`QUEUE`);
- Shared Pre-Load: can be set to pre-load data in shared memory (`PRE-LOAD`), otherwise they are stored in and loaded from shared memory at in-functor algorithms boundaries (`NO PRE-LOAD`).

Apart from Minor Blocksize, set to 1 since there is no in-functor parallelism to be exploited in this case, the optimum choice of the other parameters is not trivial and some fine tuning is needed. Figures 5 and 6 show the performance achieved in correspondence of different parameters. The best Major Blocksize values have been experimentally identified as 128 and 256, so no other values are showed in figures for brevity. It can be seen that the best configurations are the same across the two considered GPUs, but not the same across plain—ISA versions. It is also important to notice that all the available parameters can have a significant impact on performance, so they must be carefully managed. The reference implementation on Nvidia GPUs is the bitsliced AES PRNG in ECB mode described in [21] (Lim–Petzold–Koç in the following). The SubBytes phase described there is a better fit for GPUs with respect to the one contained in [1], giving a maximum performance gain of 2%. Lim–Petzold–Koç has some degrees of freedom too: blocksize and number of blocks. The optimum parameters in its case are: blocksize set to 512 on both GPUs and number of blocks set to 512 on gtx970 and to 8192 on gtx1080.

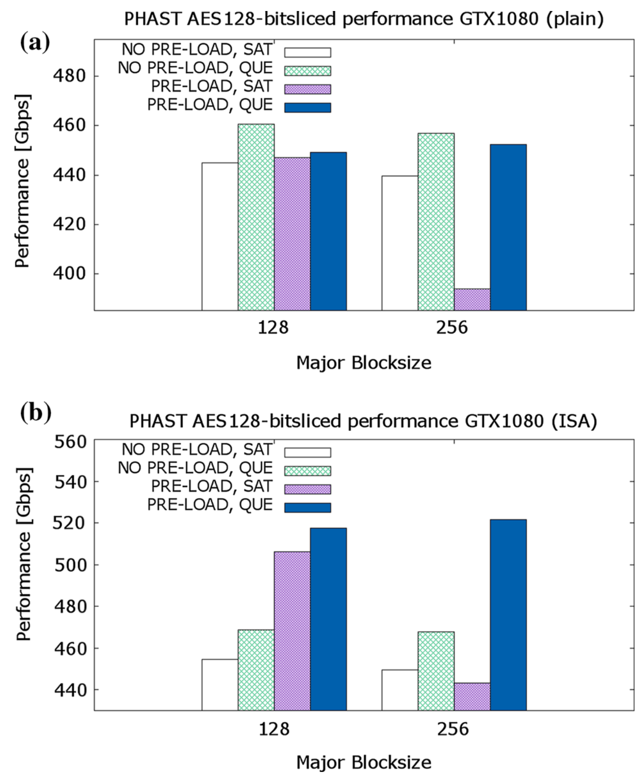

Fig. 6 AES performance on GeForce GPU GTX1080 in correspondence of different combinations of Major Blocksize, Shared Pre-Load and Scheduling Strategy. **a** GTX1080 plain version. **b** GTX1080 ISA version

Table 5 Performance comparison between AES PRNGs on Nvidia GPUs

	Lim–Petzold–Koç (Gbps)	PHAST plain (Gbps)	PHAST ISA (Gbps)
gtx970	247.31	188.57	222.72
gtx1080	570.72	452.36	521.45

Table 5 shows the achieved performance on gtx970 and gtx1080 in correspondence of the respective optimum configurations. As can be seen, the performance loss of PHAST with respect to Lim–Petzold–Koç amounts to 11.04% on gtx970 and 9.45% on gtx1080. Considering that PHAST

Table 6 Source code metrics calculated on different AES PRNG implementations: source lines of code (SLOC), Halstead's mental discriminations (H MEN D) and McCabe's total cyclomatic complexity (TOT CY)

	SLOC	H MEN D	TOT CY	Supported target
PHAST plain	475	5.990×10^6	8	CPU, GPU
PHAST ISA	500	6.984×10^6	8	CPU, GPU
no-library plain	542	9.668×10^6	11	CPU
no-library ISA	567	1.068×10^7	11	CPU
Käsper–Schwabe	226	3.412×10^5	11	CPU
assembly	8200			CPU
Lim–Petzold–Koç	839	1.281×10^7	68	GPU

code is common to both CPUs and GPUs, that it automatically accommodates different CPUs and GPUs, and that it is expressed at a higher abstraction level, such overhead does not appear to be a limiting factor in PHAST adoption.

5.3 Code metrics

Table 6 shows some metrics calculated on the source files of the different AES versions considered in this paper. SLOC is the number of source lines of code, H MEN D is the number of Halstead's mental discriminations [9] and TOT CY is the McCabe's total cyclomatic complexity [23]. Käsper–Schwabe has been split in two since no metrics can be calculated on assembly source files apart from SLOC. In the case of PHAST and no-library, both plain and ISA versions are analyzed, in order to show how much complexity increases when low-level optimizations are added. In this case, the addition of low-level optimizations acts similarly on PHAST and no-library versions, since it increases SLOC of 25 and H MEN D of 9.940×10^5 in PHAST implementation and it increases SLOC of 25 and H MEN D of 1.012×10^6 in no-library implementation.

Considering the programs where all the metrics can be calculated, both PHAST versions score the best, even better than any no-library version, which are sequential CPU-only programs. It must be also considered that both PHAST versions are compatible with multi-core processors and Nvidia GPUs, while they are being compared against platform-specific implementations. Each PHAST program can be seen, after all, as being *two programs* in one.

It is important to notice that these metrics are calculated on programs written by experts, and so the resulting differences can be considered intrinsic of the languages, APIs and libraries used. In fact, assembly is well-known for its low-level nature that inevitably leads to verbose code, and the 8200 lines of code in Käsper–Schwabe implementation are an effect of it. Analogous evaluations can be done on the many complex lines of code in Lim–Petzold–Koç implementation: CUDA has many details and many possible low-level optimizations to take care of, and all of them must be addressed to achieve cutting-edge performance. By these means, the use of PHAST library can improve programmers' productivity by allowing them to write smaller, simpler and more expressive code.

6 Conclusions

In this paper we have presented PHAST, a high-level heterogeneous C++ library that allows users to write efficient parallel code compatible with multi-core processors and Nvidia GPUs. It permits to separate application code from parameter tuning and it does not hamper the possibility to use low-level architecture-specific optimizations.

A bitsliced cache-timing-attack resistant AES-based PRNG has been implemented with PHAST library and it was compared to state-of-the-art architectural specific solutions. PHAST code resulted quite smaller and simpler than fine-tuned hand-crafted CUDA or CPU versions, since it provides high-level constructs and manages parallelization details with limited programming effort. For instance PHAST source length is 59.59% of the reference CUDA C implementation and 88.18% of the sequential C++ version for CPUs.

PHAST performance resulted comparable to native versions of the same algorithm on both architectural classes. Specifically, it delivers only a 5% overhead in case of single-threaded CPU version and about 10% for different Nvidia GPU boards. Moreover, PHAST CPU version is able to automatically scale almost linearly with the number of available cores.

Overall these results indicate that PHAST programmers can concentrate on sequential-like application code and obtain a concise parallel program that can run efficiently on both multi-core processors and Nvidia GPUs.

Acknowledgements We would like to thank Rone Kwei Lim for sharing with us the source code of his CUDA AES-based PRNG, which constituted a valuable reference for the experimental work described in this paper.

References

1. Boyar, J., Peralta, R.: A New Combinational Logic Minimization Technique with Applications to Cryptology, pp. 178–189. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-13193-6_16
2. Canright, D.: A very compact S-box for AES. In: Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems, CHES '05, pp. 441–455. Springer, Berlin (2005). https://doi.org/10.1007/11545262_32
3. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313

4. Edwards, H.C., Trott, C.R.: Kokkos: enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), pp. 18–24 (2013). https://doi.org/10.1109/XSW.2013.7
5. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, HLPP '10, pp. 5–14. ACM, New York (2010). https://doi.org/10.1145/1863482.1863487
6. Gepner, P., Kowalik, M.F.: Multi-core processors: new way to achieve high system performance. In: International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), pp. 9–13 (2006). https://doi.org/10.1109/PARELEC.2006.54
7. Gregory, K., Miller, A.: C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++. O'Reilly, Sebastopol (2012)
8. Haidl, M., Gorlatch, S.: PACXX: towards a unified programming model for programming accelerators using C++14. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM-HPC '14, pp. 1–11. IEEE Press, Piscataway (2014). https://doi.org/10.1109/LLVM-HPC.2014.9
9. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York (1977)
10. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, pp. 3:1–3:8. ACM, New York (2011). https://doi.org/10.1145/1964179.1964184
11. Hellekalek, P., Wegenkittl, S.: Empirical evidence concerning AES. ACM Trans. Model. Comput. Simul. **13**(4), 322–333 (2003). https://doi.org/10.1145/945511.945515
12. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)
13. Hunt, A., Thomas, D.: The Pragmatic Programmer. Addison-Wesley, Boston (2000)
14. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual—Volume 1: Basic Architecture. http://download.intel.com/design/processor/manuals/253665.pdf (2011). Accessed 17 Sept 2016
15. ISO: ISO/IEC 14882:2011—Information technology—Programming languages—C++. Standard, International Organization for Standardization, Geneva (2011)
16. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, CHES '09, pp. 1–17. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-04138-9_1
17. Khronos OpenCL Working Group: SYCL Provisional Specification, version 2.2. https://www.khronos.org/registry/sycl/specs/sycl-2.2.pdf (2016). Accessed 17 Sept 2016
18. Khronos OpenCL Working Group: The OpenCL Specification, version 2.2. https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf (2016). Accessed 17 Sept 2016
19. Kim, C., Burger, D., Keckler, S.W.: Nonuniform cache architectures for wire-delay dominated on-chip caches. IEEE Micro **23**(6), 99–107 (2003). https://doi.org/10.1109/MM.2003.1261393
20. Knuth, D.E.: The Art of Computer Programming. Seminumerical Algorithms, vol. 2, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
21. Lim, R.K., Petzold, L.R., Koç, Ç.K.: Bitsliced high-performance AES-ECB on GPUs. In: Ryan, A.P.Y., Naccache, D., Quisquater, J.J. (eds.) The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday, pp. 125–133. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49301-4_8
22. Lutz, K.: Boost.Compute. http://www.boost.org/doc/libs/1_61_0/libs/compute/doc/html/index.html (2016). Accessed 17 Sept 2016
23. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
24. Microsoft: Multithreading with C and Win32. https://msdn.microsoft.com/en-us/library/y6h8hye8.aspx. Accessed 17 Sept 2016
25. Miller, R., Stout, Q.F.: Algorithmic techniques for networks of processors. In: Atallah, M.J. (ed.) Algorithms and Theory of Computation Handbook, 2nd edn., Chap. 46, pp. 46:1–46:18. CRC Press, Boca Raton (1999)
26. National Institute of Standards and Technology (NIST): FIPS PUB 197: Announcing the ADVANCED ENCRYPTION STANDARD (AES). National Institute for Standards and Technology, Gaithersburg (2001)
27. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming—A POSIX Standard for Better Multiprocessing. O'Reilly, Sebastopol (1996)
28. NVIDIA: NVIDIA GF100 Whitepaper. http://www.nvidia.com/object/IO_89569.html (2010). Accessed 17 Sept 2016
29. NVIDIA: CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (2015). Accessed 17 Sept 2016
30. NVIDIA: CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2015). Accessed 17 Sept 2016
31. NVIDIA: NVIDIA GeForce GTX 1080 Whitepaper. http://international.download.nvidia.com/geforce-com/international/pdfs/geforce_gtx_1080_whitepaper_final.pdf (2016). Accessed 17 Sept 2016
32. OpenACC: OpenACC Programming and Best Practices Guide. http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf (2015). Accessed 17 Sept 2016
33. Perkins, H.: EasyCL—easy to run kernels using OpenCL. https://github.com/hughperkins/EasyCL (2016). Accessed 17 Sept 2016
34. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media Inc, Sebastopol (2007)
35. Schäling, B.: The Boost C++ Libraries, 2nd edn. XML Press, Laguna Hills (2014)
36. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL—a portable skeleton library for high-level GPU programming. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '11, pp. 1176–1182. IEEE Computer Society, Washington (2011). https://doi.org/10.1109/IPDPS.2011.269
37. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobb's J. **30**(3), 202–210 (2005)
38. Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J., Melonakos, J.: ArrayFire—a high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire (2015)