# OpenCL HLS Based Design of FPGA Accelerators for Cryptographic Primitives

We remark that this article is a preprint. The final version of this work is published in the proceedings of the 2018 International Conference on High Performance Computing & Simulation, 06th Nov. 2018. DOI:
`http://http://dx.doi.org/10.1109/HPCS.2018.00105`

**Alessandro Barenghi**[1]
alessandro.barenghi@polimi.it

**Michele Madaschi**[1]
michele.madaschi@mail.polimi.it

**Nicholas Mainardi**[1]
nicholas.mainardi@polimi.it

**Gerardo Pelosi**[1]
gerardo.pelosi@polimi.it

[1] Department of Electronics, Information and Bioengineering
Politecnico di Milano, IT

July 25, 2019

## Abstract

Modern data centers are being transformed to meet the increased processing needs of specialized workloads with an advantageous total cost of ownership. To this end, the modular design of current microservers allows the inclusion of heterogeneous computing platforms and accelerators to enhance the performance of specific workloads, while improving the power consumption and maintenance costs of the whole system. One of the fundamental application domains for datacenters is represented by bulk data encryption and decryption, as it has to be performed on the data being stored as well as on data being transmitted or received. In this paper we investigate the OpenCL programming practices to realize high-performance FPGA accelerators, thus providing a viable and more versatile alternative to the use of ad-hoc cryptographic accelerators, which are currently available in high-end server CPUs only. We validate our analysis employing AES-128 as our case study, and report energy efficiency improvements of $22.78\times$ with respect to pure software implementations of ISO standard block ciphers.

## 1 Introduction

For decades, CPU based servers have been the general-purpose workhorses of the datacentre due to the their versatility in running operations for organizations of every shape and size. Microservers are a relatively recent category of systems designed to outperform traditional (high-performance) general purpose CPU architectures in terms of energy efficiency, when carrying out specific and well-defined computing workloads. The increasing adoption of microserver based data centers has in part been supported by the growth of the web and online services, due to the predictability of the computational and I/O loads required to deliver web page elements. The quantifiable nature of the computation of the aforementioned workloads allows the hardware and software resources employed by the microserver architecture to be tailored to what is needed to execute specific tasks. The modular design of a microserver architecture allows the ef-

fective inclusion of reconfigurable and massively parallel hardware accelerators via low latency communication buses. The integration of different computing platforms allows to provide a tight match for the customers' needs. The computing platforms which may reside on a datacentre computation module range from high-performance CPUs, low-power embedded/mobile processors, low-power x86 processors, multiprocessor system-on-chips (MPSoC)s, graphics processing units (GPUs), and field-programmable gate array (FPGAs). From a data center perspective, it is of paramount importance to design and/or integrate the proper mix of heterogeneous computational modules to optimize the global efficiency of the system.

In this context, an effective development technology is represented by the Open Computing Language (OpenCL) framework [1]. OpenCL is designed to be a superset of the C99/C++11 language, allowing to target heterogeneous computing platforms, with a focus on data parallel execution. The ultimate goal of the OpenCL design is to provide seamless functional portability of an implementation of a given application across multiple computing platforms, while preserving as much as possible the high performance level attained when the application was first implemented for a given platform.

To this end, investigating the effective functional and performance portability of a given application set across substantially different computing platforms is an interesting topic for investigation, especially in the light of the recent availability of OpenCL runtimes relying on High Level Synthesis (HLS) tools to deploy OpenCL programs on FPGA based accelerators.

One of the fundamental application domains for a datacenter is represented by bulk data encryption and decryption, as it has to be performed on the data being stored (i.e., data-at-rest), as well as to ensure communication confidentiality. Such bulk data encryption and decryption is performed employing symmetric block ciphers with modes of operation providing a high degree of data-parallelism, in order to profit from the presence of dedicated accelerators. Currently, cryptographic accelerators are commonly provided in high-end server CPUs, and not in typical microserver platforms. However, microservers are usually endowed with programmable and/or reconfigurable accelerators, which support the OpenCL programming model. Parallel block cipher computation is a good fit for OpenCL realization as witnessed by the significant amount of literature on the topic [2, 3, 4, 5, 6, 7, 8, 9]. This translates to a net advantage for an OpenCL based realization in the microserver context, as it is possible to flexibly reallocate the execution of a block cipher from one accelerator to another, allowing the same platform more agility in fitting different loads. Nevertheless, we are not aware of any attempt, in the literature, to design OpenCL kernels for block ciphers targeting FPGAs (and thus leveraging HLS). Regarding the cryptographic domain, a related work is [10], which however focuses on the design of an OpenCL kernel for the SHA-1 hash algorithm.

**Contributions.** We investigate the programming practices to achieve high throughput OpenCL implementations of block ciphers targeting FPGA platforms and employing the currently available HLS tools. We analyze the OpenCL programming practices and the optimization behavior of the HLS toolchain provided by Intel/Altera, highlighting the differences from the common GPU-targeted best practices. We validate our analysis employing the AES-128 block cipher as our case study, and perform an extensive performance and energy efficiency validation on all the ISO standard block ciphers. We attain throughput improvements of one to three orders of magnitude with respect to software implementations of the same block ciphers and a $22.78\times$ average gain in energy efficiency.

## 2 Preliminaries

In this section, we provide a basic introduction to the OpenCL programming and memory models, and to FPGAs.

### 2.1 The OpenCL Framework

OpenCL [1] is a programming framework designed to address heterogeneous architectures. The main target of the OpenCL project is to provide a set of application programming interfaces (APIs) to write functionally portable code among architecturally different devices. For instance, the same OpenCL code can be compiled and run on CPUs, GPUs and FPGAs. Each OpenCL compliant device provides an OpenCL runtime library, which is in charge of deploying the code on it, as well as acting as a device driver at runtime. One of the most interesting features of the

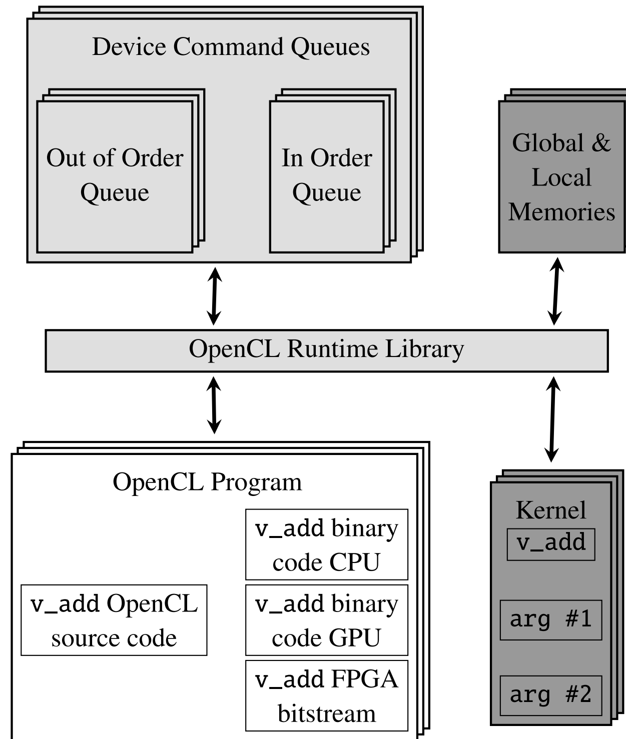


Figure 1: Logic diagram of the components of an OpenCL framework running a program with a kernel named `v_add`. Dark gray components reside on the accelerator device, light gray components run in the host main memory, the white component is the Executable and Linkable Format (ELF) *fat binary*, stored on the host. It contains the OpenCL kernel both in its source form and as pre-compiled binaries for multiple targets, and the required machine code to be run on the host and interact with the OpenCL runtime library

OpenCL runtime library lies in its ability to choose at runtime the device to run the target application on. Indeed, the OpenCL runtime must allow to run the machine code of a specific device as well as it has to embody a just-in-time (JIT) compiler to translate a source code implementation if the machine code is not already available. An OpenCL application is split into one or more *kernel* functions and some *host* code to be deployed, respectively, on one or more target accelerators and on the general purpose computational platform to which the accelerator(s) is(are) connected. The kernel functions contain the highly data parallel portion of the application, while the *host* code includes the boilerplate operations to run the kernels, transfer the data to/from the accelerator, and the code ill-fit to parallelization. Figure 1 depicts the components of the OpenCL framework involved in the execution of an OpenCL application. The OpenCL runtime library provides the functional support required to run the OpenCL program on the desired accelerator. To this end, the code of the OpenCL kernels can be provided either as a portion of source code, which is compiled for the target device at hand on the fly by the runtime, or as ready-made binary code for the said device. The OpenCL runtime library also provides data structures and accessors to communicate with the accelerator. In OpenCL, such data structures take the form of command queues, which are used by the host code to issue execution commands to the accelerator. Finally, the OpenCL runtime library provides feedback to the code running on the host concerning the correct termination of a command issued on the accelerator, and provides data transfer between the host memory, and the device memory.

**OpenCL Programming Model.** The OpenCL programming model relies on a Single Instruction Multiple Data (SIMD) paradigm, as it was designed to exploit SIMD hardware platforms (e.g., GPUs). In the OpenCL jargon, the data-parallel computation is split into basic units known as *work-item*s. A kernel is thus constituted by a number of *work-item*s known as its `global_size`. The work-items of a kernel are partitioned onto equally sized sets known as

*work-group*s. Such a partitioning aims at providing information on the spatial locality of the memory access patterns to the compiler: work-items in a work-group should work on spatially close memory locations. The number of items in a work-group is known as the `local_size` of the group.

OpenCL provides the developer with an abstract memory hierarchy to allow efficient code generation, fitting the target device memory hierarchy. The OpenCL memory hierarchy is made of four different memory classes: `global`, `local`, `private`, and `constant`. Global memory is accessible by all work-items, `local` memory is shared by all work-items in the same work-group, while `private` memory is visible only to a single work-item. These memories may be mapped onto different physical resources on the target device. The `global` memory is usually the largest and slowest one, while the `private` memory is usually the fastest and smallest one (e.g., a register file). The `constant` memory is a read-only portion of memory visible to all work-items in a kernel, and which may be mapped to a dedicated, replicated, physical volatile storage closer to the execution units, or simply in a portion of the `global` memory.

### 2.2 Preliminaries on FPGAs

FPGAs are digital devices allowing to implement a digital logic circuit by means of a set of programmable logic elements. The programmable logic elements are typically driven by Static RAM (SRAM) cells, and thus it is possible to reprogram an FPGA easily, allowing quick and effective prototyping of digital designs. The configuration process is driven by a small dedicated on-die circuit taking as input a binary sequence containing both programming commands and the contents of the SRAM elements to be configured. Such a binary sequence is known as `bitstream`. An FPGA is organized as a two-dimensional matrix of *configurable elements*. Such configurable elements are of three classes: 1) *logic block*s, which allow to implement Boolean functions and sequential components; 2) *routing block*s, which provide connections between logic blocks; 3) *I/O block*s, placed at the edges of the FPGA, which provide the means to exchange data with external devices. A logic block contains a Look-Up Table (LUT), storing the truth table of a $n$-to-$m$ Boolean function and one or more flip-flops, as well as the multiplexers to bypass them. The physical structure of routing blocks consists of a set of switches that route signals among different logic blocks. Besides the matrix of blocks, FPGAs are equipped with small ($\approx$ 16 kiB) SRAM banks, referred to as Block RAMs (BRAMs), and may also be endowed with a DRAM memory controller connected to an off-die set of DRAM banks.

**OpenCL Accelerators Based on FPGAs.** When targeting an FPGA device, the OpenCL kernel needs to be translated into a logical circuit, and encoded into a bitstream. The first step is performed via a High Level Synthesis (HLS) toolchain, which translates the OpenCL source code into an equivalent description expressed by means of a Hardware Description Language (HDL). The second step is performed via an FPGA synthesis toolchain, translating the HDL description of the circuit on a set of Boolean gates and flip flops, mapping the said circuits onto the available FPGA resources, routing the wires for the design, and finally encoding the obtained circuit into the bitstream for the target device. The aforementioned translation process is quite computationally demanding: it is thus typical for OpenCL programs targeting FPGA accelerators to include the OpenCL kernels in their bitstream form, instead of the corresponding source code. The OpenCL translation toolchain targeting FPGA-based accelerators provides effective functional portability of the OpenCL code, allowing a semantically equivalent execution on the FPGA. To this end, the HLS pass maps the global memory of OpenCL onto the off-die DRAM, which is usually available on FPGA-based accelerator cards, while it attempts to fit the local memory of OpenCL onto the FPGA BRAMs, eventually storing a small overflowing amount onto the sequential components of the logic blocks, and finally mapping the remainder onto the off-die DRAM. The private memory of OpenCL is mapped, to the best of the available resources, onto the sequential elements of the logic blocks. Despite providing functional portability, and trying to match the OpenCL expected memory latencies, the HLS synthesis tools require an appropriate coding style, which differs from the one employed for GPU targets.

Table 1: Comparison between the recommendations to apply the OpenCL best practices when considering a GPU and an FPGA device

| Programming Aspect | Programming Practice | GPU Recommendations | FPGA Recommendations |
|---|---|---|---|
| Loop Optimizations | Work size tuning | Large number of *work-groups*,*work-items* | Single *work-item*, single *work-group* |
| | Loop unrolling | Moderate tuning of unrolling factor | Accurate tuning of unrolling factor |
| Management of Addressable Caches | Store data in local memory | Fill local memory to the utmost | Fill BRAMs (i.e., local memory) to the utmost |
| | Store data in private memory | Fill private memory (registers) with restraint | Exploit FFs embedded in logic blocks (i.e., private memory) |
| | Variable/Register renaming | Automated | Manual renaming and duplication |
| Data Allocation | Memory coalescing | Devise coalescing friendly memory layout | Devise coalescing friendly memory layout |
| | Interleaved allocation | No automated support | Compiler-automated array interleaving in global memory |
| I/O latency hiding | I/O and computation interleaving | Multiple buffering | Multiple buffering, task-parallel-workers |
| Synchronization | Synchronization | `barrier`-based device-side | `cl_event`-based host-side |

## 3 OpenCL on FPGA Best Practices

In this section, we start from the programming practices delineated in the Intel technical references [11, 12, 13], to present a detailed discussion about the design strategies useful to exploit the computational power provided by FPGAs writing OpenCL kernels. Table 1 summarizes the areas and the existing practices for GPU-oriented OpenCL programming.

**Loop Optimizations.** Optimizing the shape of the loop nests explicitly present in the OpenCL kernel and the ones implicitly represented by its work-group and work-item subdivision, plays a crucial role in efficient OpenCL programming. Concerning the implicit loop nest, the recommendation for GPU and CPU programming is to design the OpenCL kernel with a large amount (i.e., tens to hundreds) of work-groups, each one composed by tens of work-items. Such a design allows the OpenCL compiler to exploit the massive parallelism exposed by the design, either directly mapping it onto the vast amount of simple computation units present in a GPU, or serializing sizable amounts of work-items to reduce thread creation costs on a CPU. In this regard the recommendation reported in the Intel guide for developing OpenCL applications on FPGA [13] is in stark contrast, and points to the design of a single work-group, single item kernel. The reason of this contrast relies in the data parallel nature of OpenCL, opposed to the so-called *pipeline-level parallelism* usually leveraged by implementations targeting FPGAs. Indeed, the Intel/Altera OpenCL compiler leverages the flexibility of the FPGA target to build heterogeneous small computation units that take care of the computation of a single work-item with a pipelined architecture. Such an architecture results in a hardware design which provides a high throughput exploiting the parallelism arising from the processing in the computation pipeline of different chunks of data that have no mutual read-after-write, write-after-write, and/or write-after-read dependencies. It is important to notice that designing an OpenCL kernel for a GPU will likely yield extremely poor performances if

targeting an FPGA, as it is limiting the amount of resources the HLS compiler has to realize the pipeline, given that multiple instances of it should fit on the same device.

Concerning the explicit loop nests in a OpenCL kernel, to exploit a pipeline-level parallelism a sufficiently long computation pipeline should be present. To this extent, performing *loop unrolling* (i.e., reducing the amount of iterations of a loop body of a factor $n$ via replication of the sequence of instructions in it $n$ times) allows the HLS compiler to build more efficient pipelines. Loop unrolling, either manual or compiler automated, yields measurable but moderate speedups on GPUs [7] as it simply saves the cost of the `jump` instructions, and is, as such, recommended as a programming technique. We note that, when targeting FPGA devices, properly tuned loop unrolling is crucial to attain a satisfactory performance level, as it directly impacts the amount of pipelining performed by the HLS compiler. In particular, the possibility of building a deep pipeline, allows to minimize memory transfers, as the inner buffers of the pipeline will be used to hold the intermediate results of the computation, which would instead be stored back to `global` memory on a GPU platform.

**Management of Addressable Caches.** Managing addressable caches, as exposed by the OpenCL `global`, `local` and `private` memories is paramount to decrease latencies. Indeed, the works on efficient GPU programming [7] stress the importance of moving the most accessed data onto the `local` and `private` memories to improve access latencies. While `local` memory management on GPU aims at filling the available one at the utmost, the `private` memory (that is a large register file; $\approx 1024$ registers) is entirely managed by the GPU compiler. To this end, the GPU programmer is advised to avoid excessive register pressure through storing in the `private` memory only a reduced amount of values.
In the case of FPGA programming, the tenant of employing to the utmost the `local` memory is still valid, as such a level of the memory hierarchy is mapped onto the available BRAMs and provides area-efficient addressing circuits with respect to storing large variables in the flip flops contained in the programmable elements. The choice of using of `private` memory when targeting FPGA accelerators differs substantially from a GPU target, as there is no pre-defined register file onto which the contents of the `private` memory are mapped. Indeed, `private` memory is mapped onto the flip flops of the FPGA fabric locating it as close as possible to the corresponding computing logic. However, `private` memory is less abundant in FPGA targets, which calls for a more careful choice of the variables to be allocated into it.
In the context of managing the `private` memory, and the variables stored thereof, it is important to notice that the register renaming performed by GPU targeting compilers may be absent in HLS tools targeting FPGAs (and indeed is absent in the Intel/Altera toolchain). Such a fact may cause a significant performance loss in case the result of an expression involving a given variable residing in private memory is stored in the variable itself. While this programming pattern is quite natural, and expected to yield an efficient translation onto a GPU target, the HLS compiler may not be able to lower such a pattern onto an efficient circuit. Indeed, such a pattern may force the HLS tool to insert a synchronous element between the computation and the writeback onto the original variable, which may have been mapped simply onto a set of combinatorial signals. The computed result is then mapped back onto the variable signals, in alternating clock cycles. To prevent such a fact from happening, inserting manually a temporary value onto which the result is saved will suggest to the HLS compiler the optimal translation strategy where the temporary value will act as a pipeline buffer.

**Data Allocation.** Allocating data in `global` memory with a layout allowing for burst transfers, also known as *memory access coalescing* should be performed in OpenCL programming as the access to `global` memory is typically performed with very wide (256-bit or more) buses. When targeting GPUs, memory access coalescing has a significant impact on performances, due to their computation paradigm, which can be thought of as a very large width vector processor. Indeed, accessing in a coalesced fashion to a portion of an array containing the input data on a GPU reduces the amount of memory transfers by a factor depending on the number of elements of the array which are loaded at once. While an FPGA-based accelerator may not leverage directly the data parallelism stemming from coalesced

memory accesses, translating it into a parallel processing of multiple data loaded in a single communication with the memory is still beneficial to the computation efficiency.

A data layout strategy which is beneficial for improving the performances of a circuit deployed on an FPGA-based accelerator is the so called *Interleaved Allocation*. Such a strategy is usually available as an automated code transformation in HLS compilers for OpenCL. The transformation interleaves the storage of the components of two arrays to optimize the transfer on-die of the values of both the operands of an expression, which should be computed element-wise on large arrays. This, in turn, allows the constant feeding of the computation pipeline implementing the OpenCL single work-item/work-group in a fashion that better fits the parallelism offered by the pipeline-level processing, enabling a visible performance improvement. On a GPU the interleaved data layout matches the same performance of vectorized transfers from/to coalesced stored values, and then it does not have the same relevance as programming practice.

**I/O latency hiding.** Given the fact that OpenCL accelerators are connected to the host via a high throughput, moderate latency bus (e.g., PCI-express), hiding the data transfer latencies and achieving efficient synchronization has a significant impact on performances. I/O latency hiding is commonly performed when developing towards GPU targets with a technique exploiting multiple buffers (typically two or three) stored in the `global` memory, to allow device-based computation on one of them and concurrent I/O transfers on the other(s). While double- and triple-buffering are usually sufficient to hide completely the I/O transfer latencies on a GPU target, the significant increase in computation throughput provided by an FPGA may leave the I/O as the bottleneck of the system.
We suggest to exploit multiple command queues and host-based workers (e.g., implemented as threads) to execute multiple kernels on the same FPGA, whenever the computational load imposed by one of the kernels is not sufficient to keep the accelerator busy during the memory transfers.

**Synchronization.** The OpenCL programming model allows only for intra-work-group synchronization of the computation at hand. While this synchronization is realized in a rather efficient fashion on GPU targets, (e.g., AMD GPUs have a dedicated hardware support for the OpenCL `barrier` primitive), its cost on an FPGA platform turns out to be extremely high, to the point of being highlighted as one of the prime points to be removed in optimizing the code by the Intel/Altera profiler. A way to achieve intra-work group synchronization when targeting FPGA accelerators is thus to move the synchronization point to the host-side queue management and exploit `cl_events` as signaling elements for single-work-group kernel completions.

## 4 Experimental Evaluation on AES

In this section we report the experimental validation of the described programming practices employing the Advanced Encryption Standard (AES) cipher as a case study [14] to gauge the effectiveness of each practice. Subsequently, we report the effects of applying all the practices to the entire set of ISO standard block ciphers [15, 16], which includes 9 widely employed block ciphers.

### 4.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a block cipher standardized by the U.S. National Institute of Standards and Technology (NIST) [14]. AES takes as input a 128 bits plaintext and outputs a 128 bits ciphertext. The plaintext is processed through a sequence of operations, called *round*, which involves a set of `xor`-linear combinations of the bits of the received input with themselves, a non-linear function being computed on them (implemented as a look-up table for the sake of efficiency) and a `xor` combination of the intermediate values in a round with a portion of the secret key. The number of rounds depends on the key length: we chose a key length of 128-bit among the standard ones (i.e., AES-128), which implies that 10 rounds are computed. Employing the other two key sizes (192b and 256b) increases by 2 and 4 the number of rounds. Since a block cipher acts on a small quantity of data, a strategy, known as *mode of operation* must be chosen to split a large input in chunks and encrypt them. We chose among

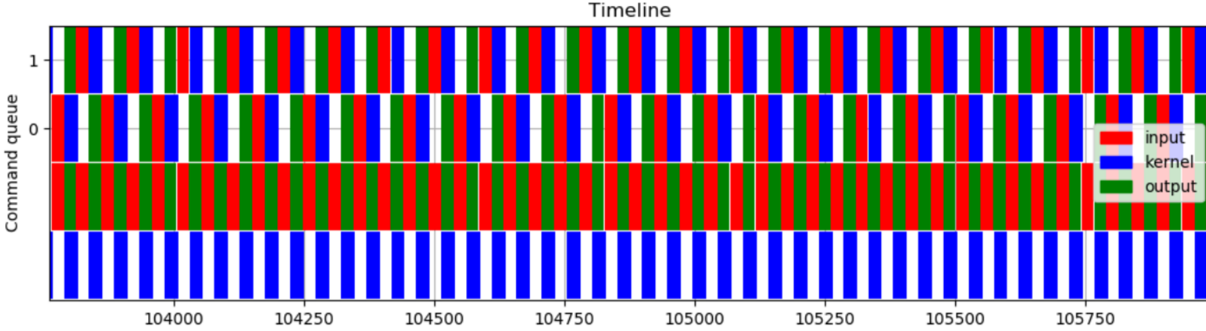


Figure 2: Depiction of the timeline of the execution of a two host threads implementation of the AES-128 ECB encyrption kernel. Each host thread is assigned to the management of a command queue, into which it issues cyclically a data transfer to the accelerator `global` memory (input, in red), a kernel computation command (kernel, in blue) and a data transfer from the accelerator `global` memory back to the host (output, green). The topmost two timelines depict which task is in execution at any given time instant in the two OpenCL command queues, while the bottom two timelines provide a synthetic view of the juxtaposition of the I/O actions alone (third timeline from the top) and the kernel computation ones (bottom timeline)

the existing modes of operation, the ones which have data dependencies allowing SIMD parallelization, namely the Electronic Codebook (ECB), the Counter Mode (CTR) and the `xor`-encryption-`xor` (XEX) tweaked-codebook mode with ciphertext stealing (XTS). All the experiments reported in this section consider the ECB mode, which basically encrypts each block with AES and then simply concatenates them. Results obtained with the CTR and the XTS mode of operations show performance results similar to the ones reported for the ECB, and consequentially do not offer further insights.

### 4.2 Programming practices evaluation

Our testbed is constituted by a dual Intel Xeon E5-2620v4 host with 128GiB DDR4 running Gentoo Linux 17.0. The FPGA based OpenCL accelerator is an *Attila* board by Reflex (RXCA10X115PF40-IDK00$A$) [17], based on the Altera Arria 10 GX (model endowed with 1150k programmable elements). We used the Quartus HLS/synthesis toolchain version 16.1.0 Build 196. We performed our tests with a set of increasing input sizes, finding out that the maximum throughput is achieved with $\approx$ 80MiB inputs, although around 10 MiB inputs are sufficient to reach more than 90% of it. We considered the sustained encryption throughput as our figure of performance, while we considered the occupied area as the number of programmable blocks over the 427k available ones on the Arria 10 GX. We report, together with the throughput figures, the area-time product as a measure of the efficiency of the solution, and the maximum frequency of the design as an additional datum hinting at the size of the longest pipeline stage. We computed the area-time product multiplying the reciprocal of the throughput by the number of occupied programmable blocks. Table 2 reports the results of the application of the programming practices described in the previous section, starting from a baseline AES-128 implementation employing a single *T-table* [14]. Such an implementation proved to be more efficient than one among the one endowed with 4 *T-table*s (i.e., the high performance software one included in OpenSSL) as the 4 tables are equivalent up to a bytewise rotation of their elements. The baseline implementation of AES achieves a significantly low overall throughput ($\approx$ 3.6 MB/s), despite the fact that the working frequency of the design is high 207.25 Mhz. We observed that such a behavior was due to the artificial insertion of additional FFs by the HLS compiler, which in turn yielded a significantly high number of clock cycles to compute a single AES. We thus tackled this issue, as described in Section 3, by manual duplication of the variables, attaining a 74$\times$ gain in the throughput.

Building on this implementation, we evaluated the effect of unrolling the loop computing the AES rounds both fully (as it is implemented in OpenSSL), and by a factor of two only. The full unrolling of the round loop has a detrimental

Table 2: Experimental evaluation of the effectiveness of the proposed programming practices. Results reported are obtained on a dual Xeon 2620 v4, with 128GiB DDR4-1600. We also report the maximum attained throughput with a strawman kernel which only performs a copy between input and output

| **Programming Practice** | **Throughput (MB/s)** | **Frequency (MHz)** | **Area-Time Product** |
|---|---|---|---|
| Baseline | 3.60 | 207.25 | 29,189 |
| Duplicated state variable | 269.38 | 238.6 | 326 |
| Full rounds loop unroll | 38.37 | 115.67 | 9,219 |
| Partial rounds loop unroll | 430.40 | 224.16 | 290 |
| key in `private` memory | 961.25 | 267.52 | 110 |
| 2 Host threads | 1,458.67 | 249.25 | 87 |
| Double buffering | 1,496.53 | 254.84 | 69 |
| No Encryption (copy only) | 1,491.90 | – | – |

effect on the performances, reducing the throughput by a factor of $7\times$. Such a performance loss is to be ascribed to the significant increase in the amount of resources used on the FPGA, in order to implement the longer pipeline allowed by the complete loop unrolling. While this in principle should result in a better exploitation of the pipeline-level parallelism, the advantages of such a deep pipeline are counteracted by an increase in the routing congestion on the FPGA, as witnessed by a sharp reduction (around $2\times$) in the maximum working frequency. By contrast, performing a manual unrolling of only two iterations of the round loop of AES-128 results in a $1.59\times$ increase in the overall throughput, as the benefits of the loop unrolling are properly reaped.

Tackling the management of the addressable caches, we evaluate the effects of storing the secret key values in the `private` memory, instead of the `local` memory where it was stored in the previous designs. We note that storing the secret key in `global` memory makes no practical sense due to the extremely high amount of accesses to it that must be done. Moving the key into `private` memory provides a significant throughput increase $2.23\times$ as a result of the key being stored into the programmable elements FFs instead of into a BRAM. Indeed, such a storage solution improves significantly the width of the access interface to the key value: a BRAM is indeed endowed with only 2 read ports, forcing the accesses to the key values to be serialized through them. Storing the key into the programmable elements FFs instead provides serialization-free access to all the portions of the computing circuit. We note that such an effect is peculiar of the OpenCL programming for an FPGA accelerator: indeed, the access latency to `local` memory in GPU is comparable to the one of the register file holding the `private` memory.

Moving onto the optimization of the input data allocation to allow coalesced memory access, we note that the encryption computation we are performing has a linear access pattern striding along a single input array (the one containing the plaintext), which is thus already in its most efficiently accessible layout. To this end, no further data allocation optimizations can be performed.

The last sets of programming practices to be evaluated include the interleaving of computation and I/O communications between host and device, and managing the synchronizations. Concerning the I/O latency hiding, we implemented both an approach relying on a straightforward *double buffering* technique, and one relying on *task-parallel workers* with two separate command queues where we alternated the kernel execution commands employing two host-side threads. Both approaches proved their effectiveness, showing a gain in throughput of around $1.5\times$ with respect to the previously achieved one, attaining $\approx 1.45$ GB/s of encryption throughput. Between the two solutions, the one relying on a single host-side queue and thread, issuing the computation commands on a buffer, while performing I/O on the other shows a slight advantage over the other. We ascribe such an advantage to the smaller host-side overhead of the implementation with respect to the two-thread approach.

As the final step of the evaluation of the effectiveness of the presented programming practices, we investigated how far the multiple buffering approach can be taken before the computation time bottleneck prevents further improvements. First of all, we determined, by benchmarking the throughput attained by a strawman OpenCL kernel which performs

Table 3: Performance evaluation of OpenCL implementations of all the ISO standard block ciphers [15, 16] on a dual Xeon 2620 v4, with 128GiB DDR4-1600. The reported area-time product is obtained considering the amount of programmable elements in the target FPGA

| ISO Std. | Block Cipher | Cipher Structure | Frequency (MHz) | Throughput (MB/s) | A×T Prod. |
|---|---|---|---|---|---|
| [15] | MISTY1 | Feistel | 258.1 | 1,503.7 | 40 |
| | HIGHT | Feistel | 189.0 | 1,360.6 | 45 |
| | DES | Feistel | 231.8 | 1,480.6 | 58 |
| | CAST5 | Feistel | 268.6 | 1,496.7 | 62 |
| | AES | SPN | 254.8 | 1,496.5 | 69 |
| | SEED | Feistel | 201.2 | 1,493.1 | 163 |
| | Camellia | Feistel | 200.9 | 1,477.7 | 204 |
| [16] | Present | SPN | 247.2 | 1,444.6 | 50 |
| | CLEFIA | Feistel | 177.5 | 1,499.4 | 203 |

a simple copy between input and output, without any further computation, that a double buffering approach is bottlenecked by the transfer capability of our setup. The throughput of the strawman kernel is substantially identical to the one of the double buffering pointing strongly to the I/O bound nature of the implementation when running on our setup (see last row in Table 2). Due to limitations of the current driver provided by Intel for our dual Xeon 2620 v4 board, (which allows only a half-duplex data transfer between the accelerator board and the host device) applying a *triple buffering* strategy yields performances substantially identical to double buffering ones, due to the serialization of the data transfer phases to/from the FPGA accelerator. Given this technical hindrance to a direct measurement of the effectiveness of a triple buffering approach in our setup, we quantified the amount of latency which can be still reduced starting from the double buffering approach. Figure 2 reports an execution timeline of the two host threads implementation of the AES-128 (first two timelines from top), highlighting which phase of the execution among copying to the device (input, red in figure), executing the kernel (kernel, blue in figure) and copying back the results to the host (output, green in figure) is being executed by the thread. The last two timelines report the juxtaposition of the input and output operations and the kernel computations of both OpenCL queues. As it can be seen, the kernel execution takes substantially the same amount of time of a single data transfer (either to or from the accelerator). It can thus be deduced that, given a driver exploiting the full duplex communication capabilities of the PCI-Express bus, a 2× improvement in the overall throughput of the implementation can be achieved employing a triple buffering strategy.

### 4.3 Performance and Energy Efficiency Evaluation on ISO Standard Block Ciphers

We now evaluate the effectiveness of the described programming practices applying all of them to the implementation of the entire set of ISO-Standard block ciphers [15, 16]. The aim of such an evaluation is to test whether the said programming practices are still valid across a variety of symmetric encryption algorithms. In particular, the nine ISO standard block ciphers can be partitioned according to their structure: either a Substitution-Permutation Network (SPN), or a Feistel network are employed to define the cipher round itself. In a SPN design, the cipher performs a computation of the same complexity on the entire cipher state, while a Feistel network cipher applies a complex function only to a half of the state, and combines via bitwise `xor` the result with the other half. As a result, Feistel network based ciphers tend to have a higher number of rounds with respect to SPNs, in turn impacting the effectiveness in pipelining and the wire routing complexity. Table 3 reports the results of our experimental campaign on the same dual Xeon 2620 v4 machine which was employed to perform the programming practice validation.

The results reported in Table 3 show that all block cipher implementations but HIGHT attain the maximum possible throughput given the capabilities of the PCI-Express bus and the Intel driver. We ascribe the reduced throughput of HIGHT to the abundant use of integer addition operations which result in longer critical paths (as indicated by the lower frequency) and the relatively high number of rounds (32). A similar effect is present in CLEFIA, which also

Table 4: Comparison of the energy efficiency of the OpenCL implementations for FPGA accelerator of the nine ISO standard block ciphers, on a Intel Xeon E5-1505M v6 CPU based host with DDR4-2133 DRAM. The CPU AES implementation employs the dedicated `AES-NI` instructions

| Block Cipher | Platform | Throughput (MB/s) | Power (W) | Energy Efficiency (MB/J) |
|---|---|---|---|---|
| AES | FPGA | 1020.8 | 15.02 | 67.96 |
| | `AES-NI` | 1678.4 | 25.79 | 65.09 |
| HIGHT | FPGA | 824.0 | 21.87 | 37.68 |
| | CPU | 20.0 | 26.01 | 0.77 |
| DES | FPGA | 764.8 | 12.62 | 60.61 |
| | CPU | 85.0 | 22.24 | 3.82 |
| CAST5 | FPGA | 814.2 | 13.63 | 59.75 |
| | CPU | 132.0 | 23.14 | 5.70 |
| SEED | FPGA | 930.8 | 14.49 | 64.23 |
| | CPU | 102.3 | 22.46 | 4.54 |
| Camellia | FPGA | 957.6 | 15.06 | 63.59 |
| | CPU | 198.0 | 22.68 | 8.73 |
| MISTY1 | FPGA | 1006.0 | 25.59 | 39.31 |
| | CPU | 22.0 | 26.07 | 0.84 |
| CLEFIA | FPGA | 1202.0 | 22.60 | 53.19 |
| | CPU | 3.0 | 26.38 | 0.11 |
| Present | FPGA | 979.0 | 25.57 | 38.29 |
| | CPU | 11.2 | 23.87 | 0.46 |

employs integer additions, although the more compact nature of the cipher does not push the FPGA resources to the point where the total throughput changes noticeably.

Having validated the proposed implementations in terms of performances, we also investigate whether employing an OpenCL HLS toolchain is able to produce accelerated implementations reaping the energy efficiency benefits of running on an FPGA. To this extent, we employ a different host, which is equipped with a dedicated power consumption monitor, on the motherboard. In particular, the host is equipped with a power-efficient Intel Xeon E5-1505M v6 CPUs and 32 GB of DDR4-2133 RAM and runs CentOS Linux 7 x86_64. Table 4 reports the results of the evaluation of the power efficiency on all the nine ISO standard block ciphers obtained with a kernel designed to be lasting long enough for an accurate power consumption measurement to be collected. The reported results show that our OpenCL implementations are able to achieve speedups between $10\times$ and $1000\times$ over a software implementation of the same block cipher. In particular, we report that, whenever available, the production grade implementations of the block ciphers present in OpenSSL were employed as the reference ones running on the CPU. It is also noteworthy the fact that our FPGA based implementation achieves a throughput comparable with the one of the dedicated ISA extension `AES-NI` of the Intel Xeon, which computes an AES encryption in 10 (multi-cycle) instructions. Indeed, given the potential advantages of a full-duplex PCI-Express driver, our implementation should outperform `AES-NI`.
From an energy efficiency standpoint, we evaluate the amount of GB per Joule encrypted by both FPGA-accelerated implementations and their software alternative (see last column in Table 4). We note that our implementation outperforms all the alternative solutions, including `AES-NI` in terms of energy efficiency. In particular we achieve a geometric mean gain of $22.78\times$ over all the block ciphers which have no dedicated instructions (the arithmetic mean gain is $79\times$), effectively proving the energy efficiency advantage of our solution.

## 5 Conclusion

We analyzed the programming practices to exploit the available HLS toolchains to deploy OpenCL programs onto FPGA based accelerators. We delineated the differences from the common GPU oriented programming practices and highlighted the impact of tailoring the code for execution on FPGA. We experimentally validated the effectiveness of such programming practices employing the AES block cipher as our case study, showing that we were able to achieve throughputs bound only by the I/O capabilities of the bus connecting the accelerator to the host. We also report energy efficiency results on the implementation of the nine ISO standard block ciphers showing a geometric mean improvement of $22.78\times$ in the amount of encrypted data per Joule spent.

## Acknowledgment

This work was supported in part by the European Community under grant agreement no. 688201 ($M^2$DC – EU H2020).

## References

[1] *The OpenCL Specification version 1.1*, Khronos OpenCL Working Group, 2011. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf

[2] A. Di Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161242

[3] G. Agosta, A. Barenghi, F. De Santis, A. Di Biagio, and G. Pelosi, "Fast Disk Encryption through GPGPU Acceleration," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2009, Higashi Hiroshima, Japan, 8-11 December 2009*. IEEE Computer Society, 2009, pp. 102–109. [Online]. Available: https://doi.org/10.1109/PDCAT.2009.72

[4] G. Agosta, A. Barenghi, F. De Santis, and G. Pelosi, "Record Setting Software Implementation of DES Using CUDA," in *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010*, S. Latifi, Ed. IEEE Computer Society, 2010, pp. 748–755. [Online]. Available: https://doi.org/10.1109/ITNG.2010.43

[5] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and Analysis of AES Encryption on GPU," in *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, G. Min, J. Hu, L. C. Liu, L. T. Yang, S. Seelam, and L. Lefèvre, Eds. IEEE Computer Society, 2012, pp. 843–848. [Online]. Available: https://doi.org/10.1109/HPCC.2012.119

[6] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "Towards Transparently Tackling Functionality and Performance Issues across Different OpenCL Platforms," in *Second International Symposium on Computing and Networking, CANDAR 2014, Shizuoka, Japan, December 10-12, 2014*. IEEE Computer Society, 2014, pp. 130–136. [Online]. Available: https://doi.org/10.1109/CANDAR.2014.53

[7] G. Agosta, A. Barenghi, A. Di Federico, and G. Pelosi, "Opencl performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 14, pp. 3633–3660, 2015. [Online]. Available: https://doi.org/10.1002/cpe.3358

[8] S. Fazackerley, S. M. McAvoy, and R. Lawrence, "GPU accelerated AES-CBC for database applications," in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March*

*26-30, 2012*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 873–878. [Online]. Available: http://doi.acm.org/10.1145/2245276.2245446

[9] E. Niewiadomska-Szynkiewicza, M. Marksa, J. Janturab, and M. Podbielskib, "A Hybrid CPU/GPU Cluster for Encryption and Decryption of Large Amounts of Data," *Journal of Telecommunications and Information Technology*, no. 3, pp. 32–43, 2012.

[10] F. Martinez-Vallina and S. Gilliland, "Performance optimization for a SHA-1 cryptographic workload expressed in opencl for FPGA execution," in *Proceedings of the 3rd International Workshop on OpenCL, IWOCL 2015, Palo Alto, California, USA, May 12-13, 2015*, S. McIntosh-Smith and B. Bergen, Eds. ACM, 2015, p. 7:1. [Online]. Available: http://doi.acm.org/10.1145/2791321.2791328

[11] *OpenCL on FPGAs for GPU Programmers*, Altera - Intel Corp., 2018. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf

[12] *Intel® FPGA Software Development Kit for OpenCL™ Pro Edition, Programming Guide*, Intel Corporation, May 2018. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[13] *Intel® FPGA Software Development Kit for OpenCL™ Pro Edition, Best Practices Guide*, Intel Corporation, May 2018. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

[14] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, ser. Information Security and Cryptography. Springer, 2002. [Online]. Available: https://doi.org/10.1007/978-3-662-04722-4

[15] ISO/IEC JTC 1/SC 27, *ISO/IEC 18033-3:2010: Information technology. Security techniques – Encryption algorithms – Part 3: Block ciphers*.

[16] ——, *ISO/IEC 29192-2:2012: Information technology. Security techniques – Lightweight cryptography – Part 2: Block ciphers*.

[17] *Arria 10 GX FMC PCIe board, using series 10 ALTERA FPGAs*, https://www.reflexces.com/products-solutions/cots-boards/pci-express/arria-10-gx-fmc-pcie-board, REFLEX CES, May 2018.