Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Evaluating OpenCL Programming Practices for FPGAs: a case study on symmetric block ciphers

Relatore: Prof. Alessandro BARENGHI

Tesi di Laurea di:
Michele MADASCHI Matr. 853500

Anno Accademico 2016–2017

Ringraziamenti

Desidero ricordare tutti coloro che mi hanno aiutato nella stesura della tesi con suggerimenti, critiche ed osservazioni: a loro va la mia gratitudine, anche se a me spetta la responsabilità per ogni errore contenuto in questa tesi.

Ringrazio innanzitutto il professor Barenghi, Relatore, il cui supporto, guida sapiente e pazienza hanno reso possibile questa tesi.

Proseguo ringraziando il personale del Politecnico di Milano, che ha fornito le infrastrutture di calcolo e facilitato le ricerche confluite in questa tesi.

Vorrei infine ringraziare le persone a me più care: i miei amici e la mia famiglia, che mi hanno sempre aiutato e sostenuto nello svolgimento del mio lavoro.

Sommario

Il panorama dei moderni sistemi di calcolo sta mostrando chiari segni di spostamento dal tradizionale, CPU-centrico modello verso architetture eterogenee, caratterizzate allo stesso tempo da elevato parallelismo ed efficienza energetica; la recente introduzione di dispositivi Field-Programmable Gate Array (FPGA) con supporto per applicazioni OpenCL è un tentativo di occupare questa posizione. Tuttavia, la natura multi-piattaforma di OpenCL rende necessario lo studio e lo sviluppo di speciali tecniche di ottimizzazione, tecniche che variano grandemente a seconda del dispositivo in esame. Relativamente a precedenti architetture con capacità OpenCL, come le GPU, gli sforzi accademici nell'esplorazione di tecniche di programmazione per OpenCL su FPGA sono agli inizi. Nel nostro scritto, studieremo la progettazione e l'implementazione di una serie di cifrari simmetrici a blocchi, e mostreremo come differenti scelte nello sviluppo di varie componenti dell'applicazione OpenCL influenzano la performance della rispettiva implementazione. Le tecniche studiate comprendono modifiche atte a ridurre il tempo di calcolo, migliorare l'uso della memoria e rendere i trasferimenti di memoria più efficienti. Il rendimento è misurato usando metriche come il throughput, il consumo di area, il tempo/memoria consumato in fase di compilazione e vari indicatori ottenuto per mezzo di analisi statica. Nel corso dei nostri esperimenti, abbiamo verificato come il loop unrolling (sia automatico che manuale) influisce positivamente sulla performance osservata; si sono anche dimostrate utili certe ottimizzazioni che impiegano speciali classi di memoria OpenCL, come la memoria locale e privata. Abbiamo inoltre osservato come l'uso delle tecniche precedentemente descritte incorre in un determinato costo in termini di risorse sulla FPGA. Abbiamo verificato che, nel particolare caso dei cifrari simmetrici a blocchi, non vi è necessità di adottare schemi particolari per accedere efficientemente agli oggetti in memoria globale. Infine, ci siamo confrontati con le limitazioni introdotte dal canale di trasmissione tra il sistema centrale e il dispositivo acceleratore OpenCL, e abbiamo misurato come uno schema di double-buffering permette di lavorare su queste limitazioni,

in modo da raggiungere un uso efficiente del canale senza ulteriori costi di area occupata su FPGA. Grazie ai dati raccolti nel benchmark finale dei cifrari a blocchi da noi implementati, abbiamo estrapolato una serie speculativa di limiti superiori per il throughput, in modo da dare uno sguardo al rendimento che potremmo potenzialmente ottenere su un sistema non affetto dalle sopramenzionate limitazioni nel canale.

Abstract

The landscape of the modern computing world is showing clear signs of a shift from the traditional, CPU-based model towards heterogeneous architectures which are at the same time heavily parallel and energy efficient; the recent introduction of Field-Programmable Gate Array (FPGA) devices capable of running OpenCL applications is an attempt to fill this particular position. However, the platform-independent nature of OpenCL makes it necessary to study and develop special optimization techniques which vary largely depending on the target device. Relatively to older architectures, such as GPUs, the academic effort to explore useful OpenCL programming practices for FPGAs is at its beginning. In our paper, we will study the design and implementation of a series of symmetric block ciphers, showing how different design choices in the various components of the OpenCL application influence the performance of the corresponding implementation. The practices studied encompass modifications which reduce the computation time, improve the memory usage and make memory transfers more efficient. The performance is gauged using metric such as throughput, area usage, compilation time/memory consumption and various indicators obtained via static analysis. During the course of our experiments, we verified how loop unrolling (both automatic and manual) positively affects the measured performance; optimizations making use of special OpenCL memory classes, such as local and private memory, did also prove very useful. We observed how using the techniques described above incurs in a certain cost in term of FPGA resources consumption. We verified that, in the particular case of symmetric block ciphers, there is no need to adopt special patterns in order to efficiently access global memory objects. Ultimately, we came to terms with the limitations created by the transmission channel between the main system and the OpenCL accelerator device, and we measured how a double-buffering scheme allows to work with those limitations, achieving efficient channel usage with no extra cost in terms of FPGA area. Using the data gathered in the final benchmark of our block cipher implementations, we extrapolated a series of specu-

lative throughput upper-bounds, thus getting a glimpse of the potential performance on a system not affected by the channel limitations described above.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Even in recent times, the ever increasing demand for more and more powerful computing systems shows no signs of weakening. The same can not be said for Moore's law, which has recently begun to manifest a significant slowdown, thus showing the limits of standard general purpose computing systems. The world of information technology promptly reacted to this change, adopting a strategy towards heterogeneous, massively parallel computing systems. One of the most important steps in this direction was the GPGPU, General-Purpose computation on Graphical Processing Units, exploiting the huge amount of parallel execution units which GPUs offer, along with a high-bandwidth dedicated memory. In order to tap into the power of such an architecture, various programming frameworks were also developed, first and foremost OpenCL (Open Computing Language), with has the inherent advantages of being open, royalty-free and multi-platform by design. The realm of OpenCL developement for GPU devices is now well explored and documented, in application domains ranging from image processing, to neural networks, to cryptography.
In even more recent times, the computing world is again showing signs of change, this time in the direction of systems with improved energy efficiency, a trend which superimposes with the already existing slant towards high parallelization. A particular family of general purpose computing devices is excellent in this domain: Field Programmable Gate Arrays (FPGA). The peculiarity of FPGAs is a highly reconfigurable internal structure, which is capable of implementing an arbitrary logic circuit, as defined by the user through an hardware description language, such as VHDL. FPGAs have been around for about 20 years already, but, for most of the time, have been relatively limited to the field of hardware design and prototyping; however, recent advances in both the hardware itself and the surrounding software ecosystem made possible to employ FPGAs for general purpose computation. Indeed, FPGAs manufacturers, such as Altera and Xilinx, started to release special compilers which allows the user to program their FPGA using

OpenCL, rather than hardware description languages. This allows to lower the entry barrier into the realm of FPGA computing, considering how the programming model of OpenCL is almost at the same level as that of major programming languages, such as C or C++. Hardware description languages, on the other hand, require a vastly different set of practices and patterns which don't usually have an equivalent in mainstream programming languages. Another important advantage of OpenCL for FPGAs is the ability to port existing OpenCL code, infrastructure and know-how from existing CPU and GPU OpenCL projects. This last point does, however, come with a very big caveat, which will also represent the starting point for the study presented in this paper. The OpenCL standard describes a programming model which is completely cross-platform, in theory; however, writing highly performant OpenCL code requires to make sharp platform-specific choices, which often do not translate well to different OpenCL-enabled devices [24]. With this consideration in mind, we set to explore, benchmark and catalog useful techniques which will allow for the development of efficient OpenCL programs, constructed having a FPGA target as the main focus. We restricted our studies to the domain of cryptography; specifically, we chose to implement 9 ISO standard symmetric block ciphers, with particular focus on AES. The domain of cryptographic functions is particularly interesting in the context of OpenCL FPGA development for a variety of reasons, but, at the time of writing, there haven't been many significant studies on this topic. Symmetric block ciphers are widely used today as the main workhorse of secure digital communications; their relative high throughput makes them suitable to encrypt large bulks of data, while the slower, more complex public key ciphers are commonly employed only for critical steps such as key exchange or authentication. In fact, symmetric block ciphers are designed from the ground up with an efficient hardware and/or software implementation in mind, a choice which is very relevant to our study, since we set to develop those block ciphers on an FPGA (close to hardware), using the high level programming language OpenCL (close to software). Our development strategy proceeded this way: first, we tackled specifically the AES blockcipher, carrying out a series of small, incremental improvements in order to probe possible programming and design patterns; this initial analysis is guided by the consolidated GPU OpenCL best practices, the Intel/Altera developer documentation and, most importantly, the practical results obtained by a range of analysis and benchmarking tools, parts of which were provided by the FPGA vendor and parts of which we developed ourselves during the length of this project. After we attained an acceptable performance in the AES implementation, we gathered all the design and programming practices which positively affected its success, and used them as a foundation for the development of the remaining block ciphers. At the end, we tested all the 9 symmetric ciphers, and made our final considerations.

0.1 Organization of the paper

In chapter 1, we describe the intents and outstanding characteristics of the OpenCL programming language and give a comparison of current state-of-art accelerated computing platforms. In chapter 2 we present our contribution in terms of programming practices which best suit the existing FPGA and compiler, compared to the existing OpenCL best practices (both in general and in the specific case of GPUs). The second part of chapter 2 contains a primer over the theoretical aspects of the blockciphers we decided to implement. In chapter 3, we show the effects of the various programming practices described in chapter 2: first, across multiple iteration of the AES blockcipher, last, across contemporary implementations of all the block ciphers in exam. Each step will be accompanied with benchmark data, as well as analytic data extracted from compiler-generated reports. In the last chapter, we draw the final conclusions and suggest future improvements.

Chapter 1

State of the Art

In this Chapter we describe the existing state of art technologies in the OpenCL ecosystem. The first part will give an outline of OpenCL in terms of design goals and programming model, with emphasis on the particular type of parallelism, memory hierarchy and synchronization introduced. The second part will describe two existing families of OpenCL-enabled accelerator devices: GPUs and FPGAs, with emphasis on the latter. In the last section, we compare and contrast the two classes of devices.

1.1 Overview of OpenCL

OpenCL is an open standard for general purpose parallel computing across a variety of heterogeneous CPUs, GPUs and accelerators, focusing on high-performance and portability. [16]

The architecture of a typical OpenCL application is divided into 2 parts: the *host* component, which is code running on the main CPU as a normal application, and the *kernel* component, which is code running on the accelerator device. The OpenCL standard defines both a series of API, used by the host application, and a cross-platform programming language, used to write the kernel code. In practice, the host-side API also needs runtime components, that is, drivers/modules which allow the native operating system to interface with the accelerator device, and a compiler to convert the OpenCL code into a device-specific binary.

As hinted before, OpenCL is designed with parallelism in mind, chiefly under the paradigm of data-parallelism: conceptually, partition the input data into an homogeneous grid, then process each partition independently, using a pool of concurrent workers which apply the same function on each data item. This behavior is in many ways similar to that of the *map()* operator offered by some high-level programming languages. In practice, the input data is not partitioned among different workers; instead, the workers are given an unique (set of) identification number, which they use to select a portion of the global input data to process. In OpenCL jargon, the workers are called *work-items*, executing a function known as *kernel*; the total amount of work-items is known as *global work size*, and may be organized across 1, 2 or 3 *work-dimensions.* The global work size and the number of work dimensions can be specified freely by the developer, and usually follows the structure of the underlying data to process: for instance, consider the example of an OpenCL application which computes the sum of two vectors of the same length; the said application should employ a global work size equal to the length of the vector, and a work dimension equal to 1. The totality of work-items is also partitioned into a series of *work-groups*, all work groups having the same user-defined *local work size*, which must be a divisor of the global work size. As stated before, each work-item is associated at runtime with a set of unique identifiers. Those identifiers are the global id, the local id, and the group id. The global id is a tuple which specifies the work-item's "coordinates" across the various work dimensions. The local id is an index which specifies the work-item's position inside its parent work-group. The group id instead is an index which uniquely identifies the work group in which the work item resides. In the upcoming sections, we will show how this design structure, reminiscent of the Single Instruction Multiple data paradigm, reflects well the underlying structure of certain hardware implementations, which are indeed SIMD machines.

Since the operation of issuing a kernel for execution is asynchronous (i.e. doesn't block until all work items have completed their execution), task-based parallelism can also be achieved. Task parallelism refers to the ability of executing at the same time multiple kernels performing different tasks, rather than just a homogeneous range of kernels performing the same function.

Another major peculiarity of OpenCL is the introduction of a memory hierarchy, that is a series of classes which qualify memory objects such as variables, pointers and arrays: **Global memory**, which can be read/written by all work-items across all work-groups. **Constant memory**, which is read-only, and otherwise behaves the same as the global memory. **Local memory** is a read/writeable memory which is shared by all work-items belonging to the same work-group. **Private memory** is a memory which can be read/written, but is visible only to individual work-items.

Generally, memory classes closer to the bottom of this list are faster than the ones listed above, although their hardware implementation may vary greatly among different devices. On the other hand, slower memory classes tend to be more abundant in term of capacity. Using the various memory classes allows the programmer to better exploit the different kinds of memories available on accelerator devices. Memory classes do also come with a set of special restrictions. First of all, a pointer pointing to a certain memory class can not be cast into a pointer to a different memory class. Another restriction is that the memory objects passed as arguments to a kernel function can only reside either in global or local memory.

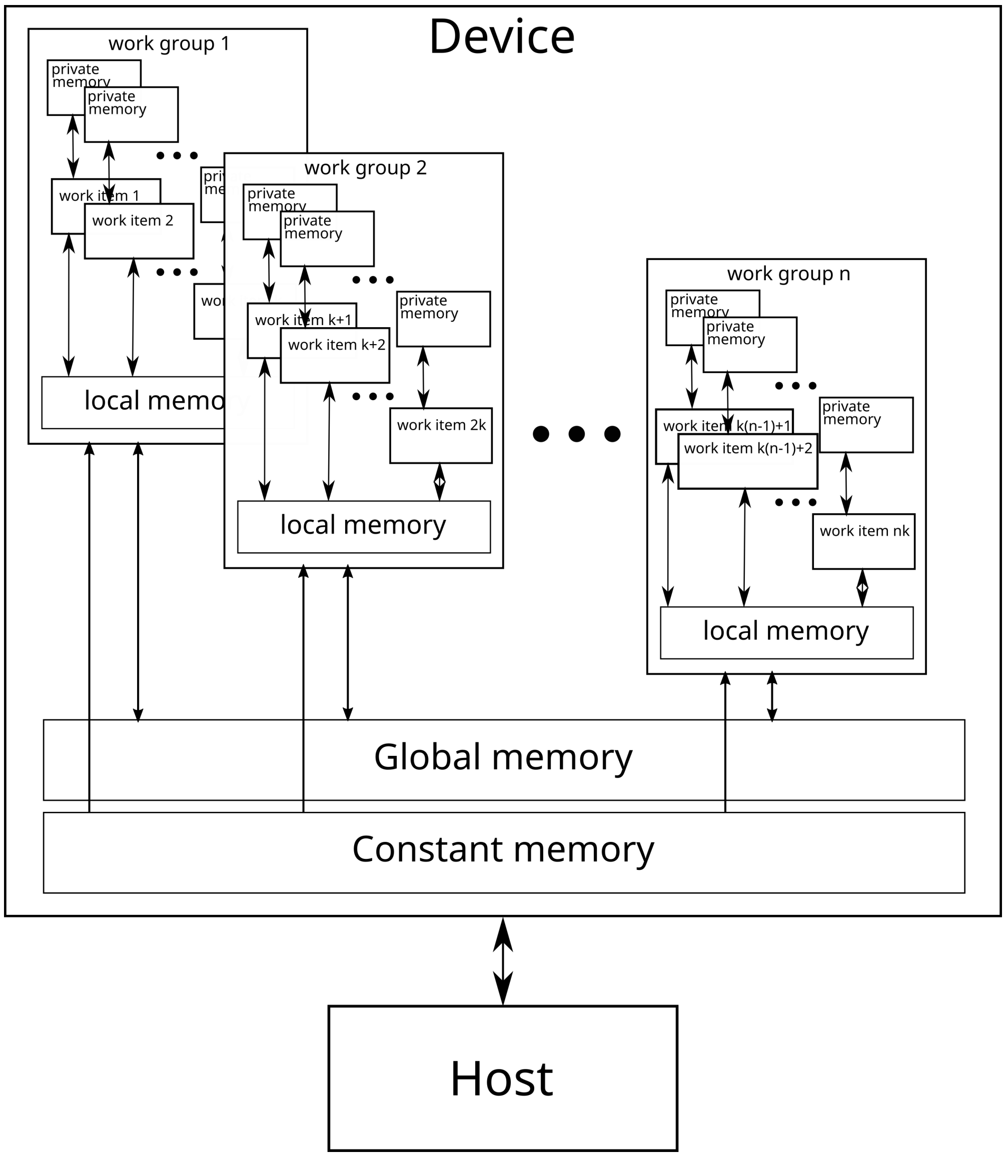

Figure 1.1: Overview of the OpenCL programming model

Figure 1.1 shows a summary of the OpenCL architecture outlined so far; the arrows denote the flow of data to/from memory.

When working with parallel/multi-threaded applications, it is important to achieve a proper inter-thread synchronization; this is handled in OpenCL via the use of special primitives, which operate at different granularity levels: Intra-work-item, using the function `mem_fence()`; it is used to make sure that all load/store operation preceding it have been properly committed to memory, so that the shared memory is viewed consistently by all the work-items concerned. Intra-work-group synchronization is instead achieved using the `barrier()` procedure, which basically blocks the program flow of the caller until all work-items in the group have reached the barrier statement. On the host side, the OpenCL API exposes a specific type, `cl_event`, which represent high level events, for example, the completion of a data transfer to the accelerator device. Chapter 3 gives an in-depth description of the various techniques and APIs to work with `cl_event` objects.

1.1.1 Comparison with C

The OpenCL language is based on C99, but comes with a set of restrictions and extensions, most notably the following ones. The C standard library is not available; the programmer may only access functions declared in their own OpenCL code, plus methods belonging to a pool of *built-in functions*. Recursion is forbidden, and will cause a compile-time error; however, this limitation can often be circumvented by slightly modifying your program. For instance, the programmer could replace a recursive algorithm with an iterative version; this modified algorithm would then simulate the recursive behavior by pushing the appropriate parameters onto an auxiliary stack data structure (also specifically implemented by the programmer). Function pointers are not allowed; while this restriction may hinder the porting of existing C code to OpenCL, it is still possible to create highly modular code through the use metaprogramming (via C macros). Finally, OpenCL does not support variadic functions or macros, that is, C functions and macros with a variable number of parameters.

On the other hand, a series of significative extensions are made to C99: special qualifiers, built-in types, built-in functions and special extensions.

OpenCL introduces a series of qualifiers which can be applied to the declarations of variables and pointers; these qualifiers are `__private`, `__local`, `__constant` and `__global`, reflecting the underlying memory hierarchy specified in the standard. Another special qualifier, `__kernel`, is used instead to identify a function as an OpenCL kernel. Finally, the special qualifier `__attribute__((attribute-list))` can be used to specify any non-standard OpenCL attributes. The `__attribute__` qualifier is actually an extension which already exists in C, although is only supported by some compilers, such as GCC (GNU Compiler Collection). An example of its use is shown in Chapter 3. The attribute qualifier can be used freely on functions, types, variables or code blocks.

The OpenCL standard also specifies a range of built-int data types; all types are made available both natively in the OpenCL language and in the API, via the inclusion of the appropriate header.

Scalar data types such as float, double, (unsigned) char / short / int / long are equivalent to the corresponding C99 types, but their bit length is fixed by the standard, whereas it is implementation-dependent in C (except certain fixed length types, such as char, float and double).

Vector data types are basically the n-component equivalent of the standard scalar types (i.e. int$n$). n can only assume values in $\{2, 3, 4, 8, 16\}$.

Miscellaneous types are opaque objects which represent structures such as images (`image2d_t`), OpenCL events (`cl_event`), memory buffers (`cl_mem`) and components of the OpenCL infrastructure (such as the `cl_program`, which represents a program containing one or more kernels; `cl_kernel`, which represents a kernel function within a program; `cl_command_queue`, which represents a channel used to send data and commands to the accelerator device).

Despite the lack of C99's stdlib, OpenCL offers various built-in functions, most notably mathematical functions, image manipulation functions, *work-item functions* and synchronization functions. Mathematical functions allow for common operations on integers, floating-point numbers and vectors, such as `abs()`, `sin()` or `distance()`. To work with the built-in image type, a family of *image functions* is provided. Common operations include getting the dimensions of an image, and reading/writing pixels at certain coordinates. Finally, the *work-item functions* are series of getters which return parameters such as the index of the current work item (`get_global_id()`), the id of the current work-group (`get_group_id()`), the total amount of work-items (`get_global_size()`), the number of work dimensions (`get_work_dim()`), and more (the full list also includes `get_group_id()`, `get_num_groups()`, `get_local_size()`).

Aside from the standard features described above, specific OpenCL implementations may also expose custom built-in functions, for instance: `cl_intel_printf`, an extension available on some Intel CPUs, allows the programmer to use the usual C `printf` statement inside the OpenCL code. Combined with the cross-platform nature of OpenCL, this allows the programmer to easily debug code targeting any platform. In the case of our project, for instance, we set up the development environment to target the main CPU instead of an FPGA, thus allowing to debug the OpenCL code very quickly and easily. Another interesting extension is `cl_intel_channels` (formerly Altera channels), which allows to use Intel channels, a feature available on some FPGAs. An Intel Channel is a blocking FIFO queue which allows different kernels running at the same time to rapidly exchange data, without accessing local memory, global memory or the host. We further investigate the usage of channels in Chapter 3.

Listing 1.1: OpenCL vector addition kernel

```
1 __kernel void vector_sum(__global const float *A,
2                          __global const float *B,
3                          __global float *C) {
4     size_t tid = get_global_id(0);
5     C[tid] = A[tid] + B[tid];
6 }
```

The snippet 1.1 above shows the OpenCL equivalent of “hello world”, a very simple self-contained OpenCL kernel performing the addition of 2 vectors, useful to demonstrate the concepts introduced before. The kernel is a normal C function (`vector_sum`), decorated with the special qualifier `__kernel` (line 1); the function parameters (pointers to float A, B and C, respectively on lines 1, 2 and 3) also need a special qualifier, which specify their memory class (`__global`). The first statement in the function's body (line 4) uses the special *work item function* `get_global_id()` to retrieve the identifier (integer) of the current work-item; the parameter (0) passed to the function is used to select which work dimension to consider. In this example, the kernel is run with a single work dimension and a global work size equal to the length of the vectors; therefore, the work item identifiers range from 0 to vector length-1. Finally, the statement at line 5 performs the actual sum, using the current work item identifier as the displacement within the vectors.

1.1.2 Host-side API

The host-side API (available for a range of programming languages, including C, C++ and Python) offers methods which handle the lifecycle of an OpenCL application.

Algorithm 1.1.1: Pseudocode for OpenCL host-side boilerplate code

```
Data: constant source, source code shown in Listing 1.1
Input: Input vector A[], Input vector B[], Integer len, equal to the length of A[],
       B[]
Output: Input vector C[] (C[] = A[] + B[])
platform ⟵ GET_PLATFORM();
device ⟵ GET_DEVICE(platform);
context ⟵ CREATE_CONTEXT(device);
command_queue ⟵ CREATE_COMMAND_QUEUE(context, device);
program ⟵ CREATE_PROGRAM(context, source);
BUILD_PROGRAM(program, device);
kernel ⟵ CREATE_KERNEL(program, "vector_sum");
buffer_A ⟵ CREATE_BUFFER(context, len);
buffer_B ⟵ CREATE_BUFFER(context, len);
buffer_C ⟵ CREATE_BUFFER(context, len);
SET_KERNEL_ARG(kernel, 0, buffer_A);
SET_KERNEL_ARG(kernel, 1, buffer_B);
SET_KERNEL_ARG(kernel, 2, buffer_C);
ENQUEUE_WRITE_BUFFER(command_queue, buffer_A, len, A[]);
ENQUEUE_WRITE_BUFFER(command_queue, buffer_B, len, B[]);
global_work_size ⟵ len;
work_dim ⟵ 1;
ENQUEUE_NDRANGE_KERNEL(command_queue, kernel, work_dim, global_work_size);

ENQUEUE_READ_BUFFER(command_queue, buffer_C, len, C[]);
RELEASE(kernel); RELEASE(program); RELEASE(command_queue);
RELEASE(buffer_A); RELEASE(buffer_B); RELEASE(buffer_C); RELEASE(context);
```

The pseudocode above (Algorithm 1.1.1) shows an example of the host side of an OpenCL application; the goal of this snippet is to drive the vector addition OpenCL kernel shown before.

The first step is to choose `cl_platform` and `cl_device`; the `cl_platform` is a vendor specific OpenCL implementation, which includes within it a list of available `cl_device`, this other object being an abstraction over the actual accelerator hardware device. In more complex applications with multiple devices and platforms, the programmer may decide to enumerate all the available devices and platforms, instead of selecting the first ones available, as we did in this example. After choosing a platform and device (lines 1-2), the next step is to use them to initialize a new `cl_context` (line 3), a global object which will hold the OpenCL envionment's internal state, and will be needed in order to create all sorts of special OpenCL objects. Once the context is ready, it can be used to initialized another important object: the `cl_command_queue` (line 4). Command queues are the communication channel used by the host to send data and orders to the accelerator; it is possible to have more than one queue active in the same context. The `cl_context` created earlier must also be used to initialize one or more `cl_program` (line 5-6); a `cl_-program` object is an abstraction over a program written in the OpenCL language, either in source form or in precompiled binary form. In order to actually use the program, one or more `cl_kernel` object (abstraction of a kernel function defined in the program) must be initialized; to do so (line 7), only the parent `cl_program` and the name of the function are needed. Before running the kernel, it needs to have the proper variables mapped to its input parameter; this is achieved with repeated calls to `clSetKernelArg()`. Handling array-type parameters is slightly more complicated. First, each one of them needs a corresponding `cl_mem` object (initialized with the required size, lines 8-10); then, the `cl_mem` buffer must be mapped to its kernel argument, as done for simple scalars (with `clSetKernelArg()`, lines 11-13); last, the input array must be copied into the accelerator device's own internal memory. The `clEnqueueWriteBuffer()` function (lines 14-15) does precisely that, it is the first of a series of functions which operate by putting a new command into the command queue. Now that the kernel is ready, it can be launched using the `clEnqueueNDRangeKernel()` call (line 18). Global work size, local work size and work dimensions are also set at this point, hence the *ND-Range* (N-Dimensional range) naming. To retrieve the kernel's output, `clEnqueueReadBuffer()` (analogous to `clEnqueueWriteBuffer()`) is employed (line 19). Finally, in order to properly shut down the OpenCL host application, the user must invoke specific destructor functions (lines 20-21) to release the OpenCL objects created so far.

Appendix A.1 shows the C equivalent of the pseudocode described in this section. In order to make the code easier to follow, many simplifications have been made: there is

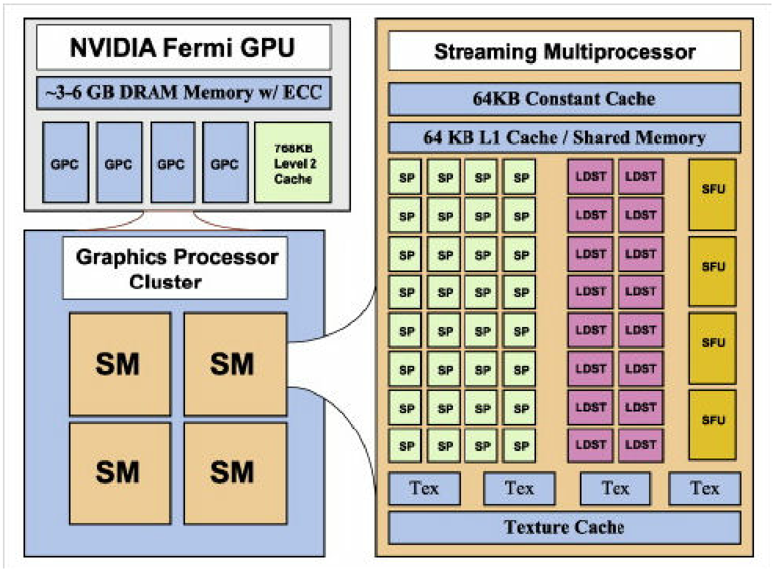

Figure 1.2: A simplified hardware block diagram for the NVIDIA "Fermi" GPU architecture, from [23]

no error checking, it is assumed to have exactly one platform with exactly one device, and no headers are included.

1.2 OpenCL-enabled accelerators

Graphics Processing Units (GPUs) are a family of specialized circuits, designed to accelerate the manipulation of images in memory. The typical operation of a GPU involves applying a specific operation (such a geometric transform, or a more complex user-defined *shader* program) to a large homogeneous matrix of data (pixel colors or vertex coordinates). This computing paradigm is more generally called *Single Instruction Multiple Data* (SIMD), and its influences are evident in the design of both the GPU hardware and the OpenCL programming model.

Figure 1.2 show a (very simplified) example of GPU structure. In the center-right of the figure, a bunch of Stream Processors (SP) are visible; SPs are the basic building blocks of the GPU. A generic SP contains a single-instruction issue unit and multiple ALU (Arithmetic Logic Unit), although the exact design varies between vendors. nVidia Stream Processors, for instance, are very simple, and only contain an integer ALU, a floating-point unit and the aforementioned issue unit. AMD Stream Processors instead err on the side of complexity, containing multiple ALUs, a self-contained register file, and a branch unit. The instruction issuing is also heavier, since the AMD SP is designed as *Very Long Instruction Word* (VLIW) machine. VLIW is a type of processing unit design, in which each instruction specifies multiple operations that the hardware is capable of executing at the same time.

Multiple SPs (usually 16 or 32) are grouped together forming a Streaming Multiprocessor (shown on the right side of Figure 1.2). A generic Streaming Multiprocessor also contains the issue logic (the same instruction is issued to all SPs simultaneously), a cache (shared among SPs), a register file and a variable number of Special Function Units, which handle particular functions specific to the graphical domain. The term "Streaming Multiprocessor" itself refers to the nVidia implementation, the AMD counterpart is called "SIMD core"; in this paper we will use these terms interchangeably.

Finally, the topmost level of the GPU chip architecture, shown in the left half of Figure 1.2, is comprised of a grid of such Streaming Multiprocessor, along with video RAM, L2 cache, and the necessary control logic.

Field-Programmable Gate Arrays (FPGAs) are a family of integrated circuits that the owner can re-configure on the fly to implement a logic-gate network of their choosing.

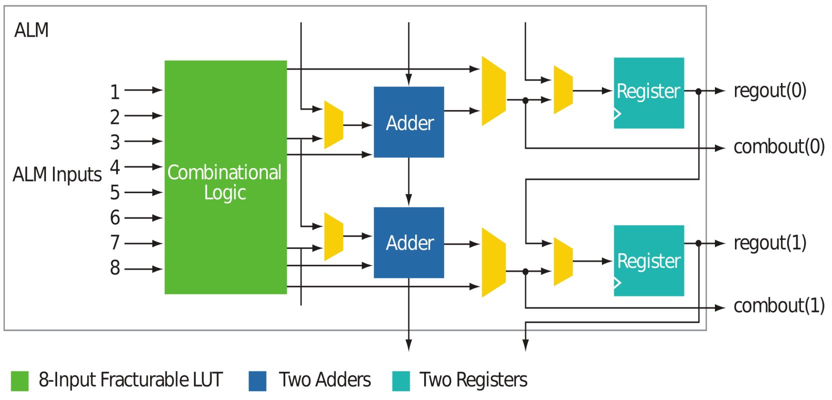

Figure 1.3: Altera's own Logic Block design (Adaptive Logic Module), from [6]. This design was employed on boards from the *Stratix* series.

Shown in Figure 1.3 is an example of *Logic Block*, the basic building block of a modern FPGA. Each *Logic Block* is a small, self-contained unit which includes: a small lookup table (LUT), one or more full-adders, one or more flip-flops, and various multiplexers and bypass paths, which allow to use just the combinatorial or just the sequential parts alone. The lookup table is programmable, and is used to implement an arbitrary combinatorial function; along with the full adders (logic circuits which perform addition of integers), it constitutes the combinatorial part of the logic block. The sequential part of the logic block is instead composed of the flip-flops. Each flip-flop can hold one bit of memory.

Multiple logic blocks are clustered together forming *Logic Array Block*s (LAB); LABs are then linked together via an “interconnect network”, which is itself re-configurable to adapt to the user’s design.

The interconnect network is built using many programmable *routing switches*, interlinked by groups of wires called *routing channels*. The whole infrastructure of LABs, switches and channels is dubbed *fabric*.

Modern manufacturers often include also special *hard blocks* into the fabric, such as RAM blocks (also known as on-chip RAM, block RAM, BRAM) and DSP blocks (used for costly floating point operations); in some cases, entire ARM processors are included [6].

An FPGA can implement purely combinatorial circuits, but is itself a clocked circuit, operating at a frequency set by an external clock. The FPGA’s routing network also handles the task of distributing the clock signal to all the required regions of the fabric; for this reason the path followed by the clock signal can variate its length depending on the particular programming being run at the moment. The external compiler, responsible for planning the board configuration, is also tasked with setting the frequency of the main clock, which must be compatible with the delays introduced by the clock paths. For a particular program, the maximum frequency at which the FPGA clock is allowed to run is dubbed *fMax* (measured in MHz).

Historically, the traditional way to program FPGAs was to use an hardware description language, such as VHDL or Verilog, but recently FPGA manufacturers started to experiment with more advanced compilers, which allow to program FPGAs using OpenCL.

The process of compiling an OpenCL program for a FPGA target is divided into two phases:

1. After a first optimization pass, the .cl code is converted to hardware description language; a report of the estimated resource consumption is also generated at this point. This phase is known as *high level synthesis*, and ranges in duration between seconds and minutes, depending on the code complexity.

2. From the hardware description code generated in step 1, the synthesis tool proceeds to create a *netlist* (a list of all the logic components and their connections). Finally,

the elements of the netlist are mapped to the physical resources of the FPGA, with a process called *place-and-route* (the placing of the logic components comes first, followed by the routing of the interconnection wires). The final result of this step is a *bitstream*, which is included in the OpenCL binary (`.aocx`). This last step lasts various hours.

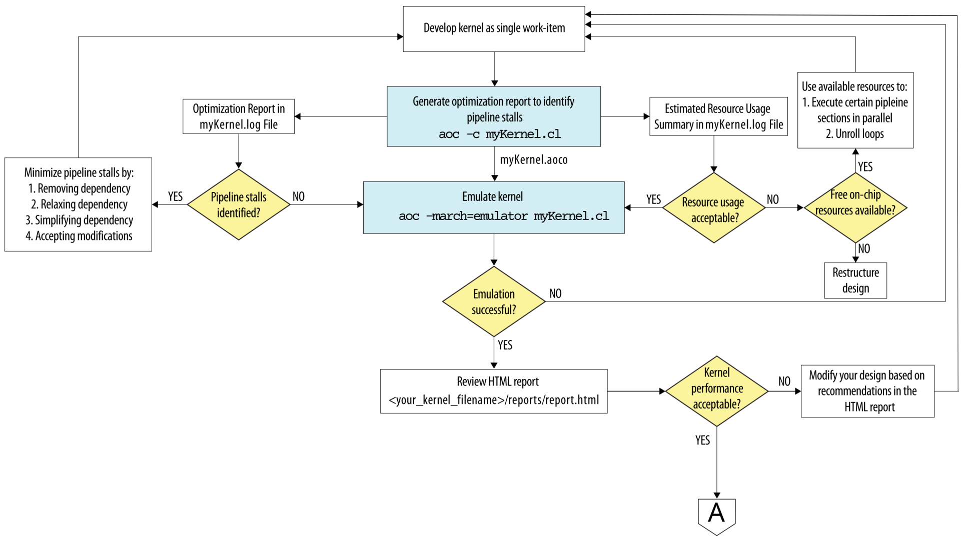
Develop kernel as single work-item
Generate optimization report to identify pipeline stalls
aoc -c myKernel.cl
Optimization Report in myKernel.log File
Estimated Resource Usage Summary in myKernel.log File
Use available resources to:
1. Execute certain pipleine sections in parallel
2. Unroll loops
myKernel.aoco
Minimize pipeline stalls by:
1. Removing dependency
2. Relaxing dependency
3. Simplifying dependency
4. Accepting modifications
YES
Pipeline stalls identified?
NO
Emulate kernel
aoc -march=emulator myKernel.cl
YES
Resource usage acceptable?
NO
Free on-chip resources available?
YES
NO
Restructure design
Emulation successful?
NO
YES
Review HTML report
<your_kernel_filename>/reports/report.html
Kernel performance acceptable?
NO
Modify your design based on recommendations in the HTML report
YES
A

Figure 1.4: Design workflow for a single-work-item kernel, from [13] (part 1 of 2)

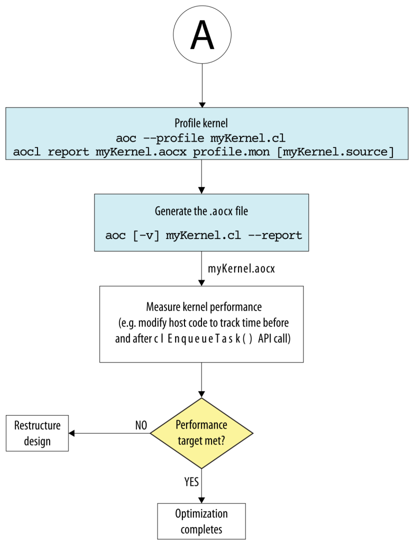
A
Profile kernel
aoc --profile myKernel.cl
aocl report myKernel.aocx profile.mon [myKernel.source]
Generate the .aocx file
aoc [-v] myKernel.cl --report
myKernel.aocx
Measure kernel performance
(e.g. modify host code to track time before and after clEnqueueTask() API call)
Restructure design
NO
Performance target met?
YES
Optimization completes

Figure 1.5: Design workflow for a single-work-item kernel, from [13] (part 2 of 2)

The flow charts 1.4 and 1.5 show how the compile steps explained above influence the design workflow of an OpenCL kernel for FPGA; steps involving the Intel compiler are highlighted in blue, while the white rectangles indicate actions taken by the developer.

The logic circuit obtained from the synthesis process does process the OpenCL work items in a way which is radically different from GPUs. Rather than building a series of functional units, the compiler instances a series of stand-alone pipelines, each processing more than one work item at a time. The Intel manual [14] [13] refers to these pipelines as *Compute units.* The Intel/Altera compiler offers an alternative to this paradigm [5], based on task parallelism instead of the usual data parallelism. Rather than having multiple "SIMD pipelines" that process multiple work items at a time, the "single work-item kernel" is implemented by pipelining only the loop iterations. (the implications and peculiarities of this technique are expanded in the following chapters).

Comparatively to GPUs, FPGAs are best suited for bit-intensive operations (`and`, `not`, `or`, `xor`, `shift`, rotation, masking) and integer operations, while floating point operations are costly. Conditional branches (which are translated to hardware paths) are also much cheaper.

There is already a wealth of studies in the realm of developing OpenCL application for FPGAs, but only a few of them are in the domain of cryptography. Among the existing studies we examined, we want to highlight [18] (study of an OpenCL implementation of SHA-1 for Xilinx FPGAs), given that has a similar domain, goal and methodology as our own study.

1.2.1 Comparative summary of OpenCL concepts and implementations

In this section, we directly compare the OpenCL devices described in this chapter. The following tables juxtapose GPU and FPGA characteristics divided in different domains: the fist tables summarizes the key OpenCL concepts introduced earlier, along with their respective GPU and FPGA realization. The second table compares price, performance and energy efficiency of various off-the-shelf GPU and FPGA models. The third table lists miscellaneous outstanding characteristics of the two device families.

Table 1.1: Quick comparison between various abstract OpenCL concepts and the underying hardware features of OpenCL accelerators.

OpenCL concept	**GPU realization**	**FPGA equivalent**
work-item	stream processor	no equivalent
work-group	SIMD core	no equivalent
private memory	register file	registers
local memory	SIMD core's internal shared memory	on-chip RAM blocks
global memory	video RAM	off-chip DDR
data parallelism	SIMD parallelism	pipeline parallelism

Table 1.1 strikes a comparison between various OpenCL concept outlined in this Chapter and their corresponding GPU and FPGA implementation. For the case of work-items and work-groups, corresponding to SPs and SIMD cores in GPU the world, we stated that there is no FPGA equivalent, which is true; however, the Intel manual hints that a Compute Unit can be considered similar to a work group, in the sense that both process multiple work items at the same time. Private memory is implemented using registers in both cases, although GPUs have a fixed register file, whereas FPGAs can implement registers on-demand by drawing from their pool of flip-flops. Local memory is also implemented in two different ways: intra-Stream Multiprocessor cache on GPUs, RAM hard-block on FPGAs. Global memory is instead implemented similarly, with some kind of large off-chip RAM in both cases. Data parallelism (final row), is also enacted differently. GPUs employ SIMD parallelism, in which multiple work-items run simultaneously on multiple processing units; on the other hand, FPGAs enact pipeline parallelism, in which multiple work-items are processed simultaneously by a pipelined compute unit.

Table 1.2: Comparison between various model of GPU and FPGA in terms of computing power, power consumption and cost. Source: [8]

Platform	**Model**	**Processing Power**	**Power Efficiency**	**Price**	**Price Efficiency**
		(TFLOPS)	(GFLOPS/W)	(€)	(€/GFLOPS)
GPU	GeForce GT 730	0.69	7	80	0.10
GPU	Radeon R9 390X	5.91	16	420	0.07
GPU	Radeon R9 Fury X	7.17	20	600	0.08
FGA	Artix-7 200T	0.65	72	190	0.29
FGA	Kintex-7 480T	1.80	72	2,500	1.39
FGA	Virtex-7 690T	3.12	78	11,200	3.59

Table 1.2 presents examples of various available GPUs (first 3 rows) and FPGAs (bottom 3 rows), along with data comparing their processing power, energy efficiency, price and price efficiency. As shown in this data, GPUs tend to have a greater processing power and cost efficiency, on the other hand FPGAs tend to have a better energy efficiency. At the time of writing, this observation remains true both for the specific examples shown above and in the general case of GPUs vs FPGAs at large.

Table 1.3: Comparison of miscellaneous special characeristics of GPUs and FPGAs.

	GPU	**FPGA**
average clock speed	1000 MHz	200 MHz
floating-point operation cost	low	high
branch operation cost	high	low
bitwise operation cost	high	low

This final table offers a comparison of various outstanding characteristics which would not fit in the previous tables. First, the average frequency at which the two classes of devices operate: around 1000 MHz for GPUs, much lower (200 MHz) for FPGAs; this big difference is also one of the main causes of the disparity in term energy consumption shown in the previous table. The other 3 rows offer a qualitative estimate of the cost (in term of both time and code/area size) of certain classes of operation. Floating-point calculations are very fast and cheap of GPU (indeed, floating point performance is one of the main concern of GPU designers); on the other hand, as stated repeatedly in the Intel manual, floating point operations are very costly on FPGA. To mitigate this, the FPGA designer usually introduce hard DPS blocks to overhaul floating point computations; the Intel OpenCL for FPGA compiler also offers special switches which simplify the implementation of floating point units, trading precision for speed. Branch operations, such as those used in loop and if statements, should generally be avoided for both classes of devices; however, FPGAs can deal with them better, by implementing the different branches as different hardware paths. On GPUs instead, the problem is usually "solved" by processing the instruction in both branches, and only retiring the instructions for the branch which is taken. Some GPUs, as in the case of AMD, are endowed with special branch units for the precise purpose of making these instruction less costly. Finally, it is important to consider also bitwise (AND, NOT, OR, XOR, NOT, bit masking and shift) operations, which take a key role in our domain of study (symmetric block ciphers). Fine-grained bitwise operation acting on some specific bits are particularly costly in GPUs (and even CPUs), which have an architecture based on fixed size words (usually 32 or 64 bit long); in order to manipulate and re-arrange specific bits, the compiler must generate multiple instructions, which the device has to execute sequentially, thus costing more memory and time. On FPGAs instead, bit operations are not a problem, since implementing basic combinatorial functions is already their normal way to operate.

Chapter 2

Performance Portability between GPUs and FPGAs

This chapter is divided into 2 parts. In the first part, we present the OpenCL design and programming best practices which we studied during the course of the project. The first section is structured in 3 parts: first, we introduce a general classification for OpenCL best practices; afterwards, we employ that classification to describe the GPU best practices which represented our initial point of reference; finally, we decline the general classification into the OpenCL FPGA best practices which we studied during the course of the project. The second part of the chapter introduces the various cryptographic concepts and algorithms which serve as a theoretical foundation for our work.

2.1 Reference: OpenCL best practices

We can divide the various OpenCL best practices into four major classes, depending on their area of influence. First, the class of techniques which allow to maximize the accelerator's resource usage in terms of **computation time**. This category includes practices such as *global work size tuning*, *local work size tuning* and *loop unrolling.* The first two techniques involve finding the optimal values for local and global work size, either by exhaustive search or through educated guesses. The last practice is instead performed by the compiler, which will attempt to unroll any loop decorated using the special directive `#pragma unroll`. The compiled code inlines all the loop iterations, thus removing the branch/control instructions handling the loop condition. The literature reports speedup of about 3.7× when unsing this class of techniques, for both GPUs [10] [20] and FPGAs [15]; furthermore, FPGAs running a single task kernel can achieve an even greater speedup of about 7× [15].

The second wide class of OpenCL best practices contains techniques used to maximize the exploitation of **memory resources** (local and private) offered by the accelerator. This is usually achieved by copying a portion of often re-used global data into a local (or private) memory buffer, processing it, and then writing it back to global memory. In the case of local memory, the transfer is split among all work items belonging to the same work group, which will then synchronize using a `barrier()` primitive; at this point, the copy operation is considered complete. Despite the apparent advantages offered by local memory, there are documented cases [27] when a OpenCL FPGA application showed little to no speedup from the introduction of local memory.

The third big category is made up of those practices which render the **access to global memory** more efficient; the techniques used to achieve this in OpenCL are collectively classified as *memory coalescing*. The basic idea behind memory coalescing is to have work items access the global memory with high locality; this can be done both within a single work item, by accessing global buffers with a simple, predictable pattern (i.e. linearly), and between different work items, by having consecutive work items access consecutive memory regions. From a developer's perspective, memory coalescing involves properly laying out the global data in memory; for example, switching between storing a matrix columns-first or rows-first. The literature reports a speedup of about 30x from improving the global memory access pattern on GPUs [26].

The fourth and final category encompasses the practices which optimize the transfer of **data to and from the host**. The essential way to achieve this is the practice of *multi-buffering*, which is mostly host-driven, and therefore equally effective for both GPUs and FPGAs. Multi buffering in OpenCL involves keeping multiple copies of each kernel, command queue and memory buffer object; the host-side application orchestrates the input/output transfers and kernel execution in order to maximize the input/output channel usage: while the kernel is busy processing the current buffer, the next one is already being independently transferred. Depending on the type of host-device channel, double-buffering or even triple-buffering can be implemented, providing an ideal speedup of 2x and 3x respectively. Another relevant practice is that of properly aligning the buffers used to copy global data to/from the accelerator. The device usually communicate with the main system memory using a Direct Memory Access (DMA) engine; however, those DMA implementations often require the DMA buffer to be properly aligned. The OpenCL runtime is usually capable of detecting misaligned buffers, and will schedule their transfer on a slow, non-DMA channel with a huge performance penalty.

2.2 Reference: OpenCL best practices for GPU

As hinted in the previous chapter, Stream Processors and Stream Multiprocessors are the GPU equivalent of the OpenCL concepts of work item and work group; the consequences of this fact are evident in all the GPU best practices. In the case of global work size tuning, GPUs favor large work sizes, to better exploit the huge number of SPs available. The process of tuning the local work size is trickier; given an architecture with 16-32 SPs per Stream Multiprocessor, the programmer should dimension their work groups as a multiple of that number. Usually, the ideal local work size lies in the interval 128-512, and is dependent on both the CPU vendor/model and on the particular kernel in use [4]. The effects of loop unrolling must be viewed in light of the internal structure of the SP, which too varies between GPU vendors. nVidia GPUs employ a simpler SP with no branch unit, which benefit greatly from loop unrolling. AMD GPUs on the other hand employ more complex SP endowed with their own branch unit, which are better able to handle branch operations, and thus see a lesser performance gain, compared to nVidia.

The bank of fast, locally shared memory included in each Stream Multiprocessor is used to implement the OpenCL concept of local memory. Private memory is instead implemented using the Stream Multiprocessor's register file, which is even faster than local memory; however given the scarcity of register resource (even scarcer considering that it must be split across all the SPs in the SM), GPU developers prefer to use only local memory for caching, and leave the private memory for simple variables. Another reasong for caching in local memory, rather than in private memory, is the so called "register spilling": when allocating a memory buffer too big to fit into the available registers, the slower global memory will be used instead. Regarding the barrier constructs, we see them implemented differently by different vendors: AMD cards possess special hardware to support synchronization; nVidia cards instead handle the synchronization completely via software.

Memory coalescing is straightforward to implement in the GPU, as the compiler can simply replace multiple separate load/store instructions with a single load/store instruction operating on larger data, thus reducing overhead. The detection of such consecutive load/store operations is performed via static analysis at compile time, but this analysis can be hindered by code which accesses the memory in convoluted ways, such as making extensive use of pointer arithmetic.

2.3 OpenCL best practices for FPGA

The OpenCL best practice for an FPGA platform differ greatly from those employed in GPUs and other families of OpenCL capable devices. This is evident from both the official Intel documentation [14] [13] and our experimental findings.

2.3.1 Work dimension tuning and loop unrolling

Our experimentations focused, since the beginning, on the developement of single-work-item kernels. This was motivated by the recommendations found in the official Intel documentation [14] [13], as well as the results of brief tests involving multi-work-item kernels. Indeed, single work item kernels allow to make better use of the pipeline-parallelism characteristic of FPGAs.

In order to be considered a *single work item kernel* by the compiler, an OpenCL function must satisfy certain prerequisites: Global work size, Local work size and Work dimension must be fixed to 1; furthermore, the kernel must not invoke any of the *work item functions* specified in the previous chapter (`get_global_id()`, `get_group_id()`, `get_global_size()`, `get_work_dim()`, `get_group_id()`, `get_num_groups()`, `get_local_size()`). Of course, such a kernel needs to receive the whole input vector and its dimension as explicit parameters.

In contrast with the radically different parallelization paradigm, loop unrolling is a technique which translates well from the GPU to the FPGA world. There are however a couple of caveats:

- For maximum performance gain, the loop should be free from loop-carried data dependencies
- Unrolling a complex loop could incur in a big cost in term of area, since the hardware implementation will duplicate the combinatorial logic used to compute the loop body.

Where, for some reasons, normal loop unrolling is not applicable, it has proven useful to manually unroll the loop by a factor 2, by halving the number of loop iteration and repeating the instruction inside the loop body twice. For even greater performance enhancements, the programmer could attempt to combine this technique with normal compiler-driven loop unrolling.

2.3.2 Local memory and synchronization

Copying the frequently re-used data to local memory works in the FPGA domain as well as it does for GPUs. Of course, when using a single work-item kernel, it must be implemented differently from the GPU variant; an easy way to do it would be to create a simple "library" function, akin to `memcpy` in standard C.

Utilizing local memory on a FPGA however yields a couple unique pitfalls; first of all, the compiler may decide to replicate the local buffer multiple times inside the Block RAMs (BRAMs), in order to decrease the kernel latency and increase the fMax (the maximum frequency at which the FPGA hardware is allowed to operate). For this reason, OpenCL program files containing more than one kernel can reach high level of block-RAM usage; the programmer could work around this issue by separating independent kernels into multiple stand-alone .cl files.

Unique to the FPGA domain is also the ability to implement local variables using flip-flop registers; on top of the general access speedup, the kernel is also made leaner and faster, because accessing the private memory in registers does not involve load or store blocks, unlike the case of local and global memory. The programmer can trigger this behavior by using the special qualifier `__attribute__((register))`. That being said, the final decision of whether to use RAMs or registers is always up to the compiler; this is somewhat similar to register spilling in GPUs, except that the data is spilled into local memory, rather than global memory.

Another FPGA peculiarity is the implementation of the `__constant` memory: constant memory is composed of a fixed-size (16KB by default, modifiable at compile time) cache located inside the block RAM, and a "master copy" located in global memory. The lookup of a constant memory object passes first through the relatively fast cache; in case of read miss, the value has to be loaded from global memory, with a big time penalty.

In regards to the use of local memory, we discovered a peculiar compiler quirk. We found that the Intel compiler tends to implement memory-intensive operation in a rather explicit way which is usually not ideal.

```
1 int state;
2 int pads[16];
3
4 for (int i=0; i<16; i++) {
5   state = state ^ pad[i];
6   state = some_function(state);
7 }
```

```
1 int state, tmp_state;
2 int pads[16];
3
4 for (int i=0; i<16; i++) {
5   tmp_state = state ^ pad[i];
6   state = some_function(tmp_state
        );
7 }
```

For instance, consider the code snippet shown above on the left: we have a variable, called `state`, which is repeatedly updated, first (line 5), by xoring with some other variable named `pad`, then (line 6), with the result of some unspecified function applied on state itself. The Intel compiler would implement line 6 in the exact way we wrote it: load `state`, load `pad[i]`, compute `state xor pad[i]`, store the result to `state`. However, the hardware is not capable of both loading and storing the same variable (state) in the same cycle; the whole pipeline will be slowed down in order to complete the load/store operations in different cycles. The issue can be fixed by slightly modifying our code, introducing temporary variables / buffers. The snippet on the right achieves exactly this, with the introduction of the extra variable `tmp_state`. In line 6, `tmp_state` is used to contain the result of `state xor pad[i]`, afterwards, state is updated with the value of some unspecified function, applied to the temporary state computed before. It's easy to see how this modified structure behaves the same as the one in the left snippet, with the important difference that `state` is no longer modified in-place. Our example was built on purpose to show this principle in action; in the real world, restructuring existing code to accomodate this compiler quirk may be more complicate to pull off.

Given the single-work-item nature of our kernel, there's not much need for inter-kernel or intra-kernel synchronization; all the synchronization steps are performed host-side. There are mainly 4 ways of doing this. First, by using the `clWaitForEvents()` function, which receives as input an array of `cl_event` objects (and its length); the function will block the program execution until all the events in the list have completed. A similar technique involves using *wait lists*, a couple of optional parameters in `clEnqueue*` functions, which specify a set of events that has to complete before the new enqueued operation can start. `clEnqueue*` functions can also optionally fire a `cl_event` on completion, thus allowing to "daisy chain" enqueued operations. The events are opaque objects, representing the state of an enqueued command; the user does not manage them directly, but instead relies on specific OpenCL API functions designed to react to specific events. An example of this technique is shown in our OpenCL host side boilerplate Listing A.1 (the relevant part has also been reported here).

Listing 2.1: fragment from OpenCL host-side boilerplate code

```
51 clEnqueueNDRangeKernel(
52     command_queue,
53     kernel,
54     work_dim,
55     NULL,
56     &global_work_size,
57     NULL, // let the implementation choose the local work size
58     0, NULL,
59     &kernel_complete);
60
61 clEnqueueReadBuffer(command_queue, buffer_C, CL_TRUE, 0,
       vector_length, C, 1, &kernel_complete, NULL);
```

The final parameter of the `clEnqueueNDRangeKernel()` call is a pointer to the event `kernel_complete`; on return, the function sets the event object to represent the execution of this particular kernel. On line 61, we see how the `clEnqueueReadBuffer()` invocation makes use of wait lists: the 8th parameter (`&kernel_complete`) is an array of events; the 7th parameter (`1`) is the length of the array. These parameters specify a list of events that must reach the complete (`CL_COMPLETE`) state before the newly enqueued operation (reading `buffer_C`) can begin. Another, easier way to achieve synchronization is to simply use *blocking operations*; `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` both expose a boolean parameter which allows the programmer to switch their behavior between blocking and non-blocking. Finally, the OpenCL command queues offer a degree of synchronization by default; all the operations inserted into the same command queue are guaranteed to be issued and completed in the same order they were enqueued by the host. This is the default behavior, but the programmer can override it by setting the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag when creating the command queue.

We have found that it's best to use more than one technique as the same time, as done in Algorithm 2.3.1 (the algorithm is explained extensively in the upcoming section), which is built around the idea of enqueuing most of the operations in a quick "burst", and only waiting for the last one to complete. There are however some small caveats to consider. In Algorithm 2.3.1, we did use multiple queues without any form of inter-queue synchronization; this is perfectly fine in our domain (implementation of the ECB, CTR and XTS modes of operation for certain block ciphers), but not in general, especially in presence of some kind of on-device state which persists across kernel executions. The command queues do also have a limited (implementation dependent) capacity, and the burst code should be built with safeties against overloading the queue; our solution was to periodically perform a blocking "synchronization step" (using `clWaitForEvents` or a blocking read). Another important fact to consider is that, as required by OpenCL, the input buffer should remain unchanged during the duration of write operation; being the write operation nonblocking, there is no way for the caller to know when they can reclaim the input buffer, save for waiting the final synchronization. In contrast to Algorithm 2.3.1, which only returns after the final synchronization with no issue in this sense, our final implementation employs primitives which process single blocks on-demand. In order to satisfy the OpenCL requirement while also keeping our flexible design, we introduced an optional callback to notify the caller when the input buffer is safe to re-use. To do so, we leveraged the method `clSetEventCallback`, which allows to associate a specified `cl_event` with a custom callback.

2.3.3 Global memory access optimization

Memory coalescing is implemented similarly on FPGAs and GPU; the main difference is that, while GPUs have discrete load/store unit (usually located inside the SM), the FPGA must instead infer load/store units using LUT and FF resources. Indeed, better memory coalescing also results in lower resource consumption.

During the course of our experiments, we were limited to the domain of block ciphers, which do already access their input and output buffers in a simple linear way, offering no room for further improvements.

What we tried was to compile the same kernel with and without memory interleaving. Memory interleaving is a feature which allows to store the input/output buffers interleaved in global memory; this would allow better memory coalescing for operation involving multiple buffers, such as addition of two vectors.

As shown in the following chapter, this feature didn't change the performance of our kernel at all.

2.3.4 Host-device transfers optimization

While working with task-parallel single-work-item kernels on FPGAs, it is critical to consider the dual concepts of multi-buffering and multiple alias kernels. Multi-buffering, as already described above, is particularly useful when the input/output transfer is the bottleneck; the multi-buffer infrastructure must manage multiple instance of the same OpenCL objects, including kernel objects pointing to the same `__kernel` function. The “multiple alias kernel” technique is instead useful when kernel execution is the bottleneck; the host-side and device-side code structures are almost the same, with a key difference. In order to use this design pattern, the programmer must explicitly declare multiple “alias” kernel function in their OpenCL code (i.e. `kernelFun_1()`, `kernelFun_2()`, ...), in order to have the compiler generate multiple coexisting pipelines; on the host-side initialization code, the various kernel objects must be initialized to point to different kernel “aliases”.

Algorithm 2.3.1: Pseudocode for host side multi-buffering

```
Data: constant $NB$, number of buffers used
Input: Input blocks $In[]$, Kernel objects $K[NB]$, Memory buffer objects
       $Mwrite[NB]$, Memory buffer objects $Mread[NB]$, Command queues
       $Q[NB]$
Output: Output blocks Out[]
1 $b \longleftarrow 0$;
2 while has new input blocks do
3     ENQUEUE_ASYNC_WRITE($Q[b]$, $next\_input$, $Mwrite[b]$);
4     SETUP_KERNEL_PARAMS($K[b]$, $Mwrite[b]$, $Mread[b]$);
5     ENQUEUE_ASYNC_EXEC($Q[b]$, $K[b]$);
6     ENQUEUE_ASYNC_READ($Q[b]$, $next\_output$, $Mread[b]$);
7     $b \longleftarrow (b + 1 \mod NB)$;
8 for $i \leftarrow 0$ to $NB$ do
9     WAIT_FOR_FINISH($Q[i]$);
```

Algorithm 2.3.1 shows an example of the host-side architecture for managing a multi-buffered infrastructure; depending on how the kernel objects $K[]$ are initialized, this same code could handle both simple multi buffering and the “kernel replication” discussed before. The algorithm is expressed in pseudocode which is close in style to the underlying

OpenCL version, but unneeded details have been abstracted away. In line 1, we initialize variable b (current buffer). For every input block available, we run an iteration of the main loop (line 2); we don't assume to know a priori the amount of blocks which the algorithm will consume. For each iteration, a series of asynchronous operations are inserted into the current command queue ($Q[b]$): the transfer of the incoming input to the device (line 3), the execution of the current kernel (line 5), the transfer of the resulting output from the device (line 6). Furthermore, buffer read/write commands also require an OpenCL memory buffer object to act as a destination/source; before running the kernel, we make sure that said buffers are correctly mapped as kernel parameters (line 4). Finally, we swap buffer (line 7) and continue looping. Once there are no more input blocks to process, we can enact the final synchronization phase: for all available buffers (line 8), we wait the corresponding command queue to complete all operations (line 9).

Due to a limitation in the board's Direct Memory Access (DMA) controller, all memory transfer to and from the host system needed to use memory buffers with a proper alignment (in our case, 64 bit). It's the programmer's duty to allocate a buffer with the proper alignment, for instance, using the `aligned_alloc()` function, defined in `stdlib.h`. Failure to properly align the buffer is not catastrophic, but the driver will be forced to use very slow non-dma transfer.

2.4 GPU and FPGA best practices compared

Table 2.1: The following table summarizes in synthetic form the FPGA and GPU best practices discussed above, comparing and contrasting equivalent techniques across the two domains.

OpenCL best practice	**GPU equivalent**	**FPGA equivalent**
Global work size tuning	Large work size	Global work size = 1
Local work size tuning	Fine-tuned local work size (multiple of 32)	Local work size = 1
Loop unrolling	Loop unrolling	Loop unrolling; *manual partial loop unrolling*
Cache data in local memory	Cache data in shared memory	Cache data in BRAMs
Cache data in private memory	Not used	Cache data in registers
No equivalent	No equivalent	*Same variable load/store quirk*
Synchronization	`barrier`-based device-side synchronization	`cl_event`-based host-side synchronization
Memory coalescing	Locality of global memory access	Locality of global memory access; compiler-driven memory buffer interleaving
Multi-buffering	Multi-buffering	Multi-buffering; *Multi-task-parallel-workers*
Host-side buffer alignment	Host-side buffer alignment	Host-side buffer alignment

The table above offers a brief summary of the best practices discussed in this chapter. Useful new practices not included in the Intel programming manual [14] [13] are listed with *emphasis*. In order, these techniques are: Manual partial loop unrolling, which we found to be very effective in practice (Altera's online discussion forums also provide anecdotal evidence in support of manual loop unrolling on FPGAs). The quirk occurring when loading and storing the same variable in the same statement is also a new finding.

Finally, the technique used to create multiple seamless workers via kernel aliasing is also a novel one, exploiting task parallelism which was itself briefly explained in the documentation. In chapter 4, we make the case for a source-to-source compiler for translating a GPU-oriented OpenCL kernel into a FPGA-oriented one, based on the similarities and differences discussed in this chapter.

2.5 Our case of study: symmetric block ciphers

The studies and experimentations conducted during our project are localized to the domain of cryptography; specifically, we set the goal of optimizing the implementation of 9 ISO standard symmetric block ciphers (AES, DES, Camellia, CAST5, CLEFIA, PRESENT, MISTY1, HIGHT and SEED). For each of those ciphers, we implement methods for handling the ECB, CTR and (when applicable) XTS mode of operation.

A symmetric encryption scheme is formally defined as a tuple

$$< \mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \{\mathbb{E}_k(), k \in \mathcal{K}\}, \{\mathbb{D}_k(), k \in \mathcal{K}\} >$$

where

$$\mathbb{E}_k : \mathcal{M} \to \mathcal{C}$$

$$\mathbb{D}_k : \mathcal{C} \to \mathcal{M}$$

In other words, we have a bijective function, called the *encryption function* ($\mathbb{E}_k()$), which maps elements from the plaintext set ($\mathcal{M}$, also known as "message space") into elements of the ciphertext set ($\mathcal{C}$, also known as "ciphertext space"), using a specific key k belonging to the set of keys, or "keyspace" ($\mathcal{K}$). The decryption function, similarly, maps elements from the ciphertext space into elements of the message space, and is also bijective. The encryption scheme is called "symmetric" when the encryption and decryption functions both use the same key, in such a way that applying the decryption function on the result of the encryption function yields the initial message ($\mathbb{D}_k(\mathbb{E}_k(m)) = m$) and vice-versa ($\mathbb{E}_k(\mathbb{D}_k(c)) = c$).

Elements of both the message space and the ciphertext space are constructed as strings of elements from the alphabet set $\mathcal{A}$. A *block cipher* is an encryption scheme which breaks the whole plaintext into *blocks* of a fixed length (*blocklength*) over the alphabet $\mathcal{A}$, and then encrypts one block at a time [19]; the block length is usually expressed in bit.

Since most real-world message exceed the *blocklength*, we need a technique to properly apply the block cipher primitive onto a larger multi-block message. The family of techniques used for this purpose is called “modes of operation”.

The simplest of all modes of operations is the **Electronic Code Book** (ECB) [19] mode. ECB works by evenly splitting the whole input message into blocks of size *blocklength*; to do so, the input message must have a total length which is multiple of *blocklength*. Once the message has been split, the encryption function processes the input blocks separately; the single output ciphertext blocks are concatenated to form the complete ciphertext. In order perform decryption, it is sufficient to swap the encryption function with the one for decryption. The fact that each encryption operation is independent from the others makes ECB trivial to parallelize, but this fact also brings some intrinsic weaknesses. First of all, identical message blocks are translated into identical ciphertext block, thus leaking information about the structure of the original plaintext, provided that the message contains many repeated block. Furthermore, ECB does not provide any data integrity checks, thus allowing the ciphertext blocks to be freely swapped, rearranged and individually modified (although the individual modification does result in a “garbled” decrypted block).

Another mode of operation we used in our project is the **Counter** (CTR) mode. To set up CTR mode, another special parameter is needed; this parameter is called Initialization Vector (IV), and is used to initialize the value of an internal variable called “counter”. The length of the IV must be equal to the *blocklength* of the blockcipher being used. The Counter Mode routine works by incrementing by one the counter variable, which is then encrypted using the blockcipher’s encryption primitive. Finally, the encrypred counter is xor-ed with a message block (the message is split into blocks in the same way as for ECB), yielding the corresponding ciphertext block. Since the counter is always incremented by one, we can compute, for the i-th block, the corresponding value of the counter as $IV + i$. Given that we can compute the value of each counter independently from the predecessor, this mode of operation can too be efficiently parallelized:

$$C_i \leftarrow \mathbb{E}_k(IV + i) \oplus M_i \tag{2.1}$$

For the i-th message block M_i and ciphertext block C_i. Another useful feature of CTR is that the Counter-mode encryption procedure is identical to the Counter-mode decryption procedure; furthermore, CTR only makes use of the blockcipher’s encryption primitive.

For this reasons, CTR is often used in contexts where hardware and memory resources as scarce, such as in embedded devices. Compared to ECB, CTR does not encrypt identical message block into identical ciphertext block, and is also immune to block rearranging; however, CTR remains unable to detect change made to single ciphertext blocks; even worse, flipping specific bits of the ciphertext will result in flipping the corresponding bits of the plaintext. Another caveat on the use of CTR is the IV choice: the IV should be chosen unpredictably (i.e. using a good random number generator), and never re-used.

Finally, we implemented also a third mode of operation: XEX-based tweaked-codebook mode with ciphertext stealing, or **XTS** [2].

To properly explain its structure, we will split the name into 2 parts, first: "XEX-based tweaked-codebook" [22]. The structure of a XEX-based tweaked-codebook superficially resemples that of CTR, in the sense that an initialization value is used to generate a series of sequential pads, which are then merged in some way with the plaintext blocks. To do so, XEX maintains an internal variable, called "tweak"; the tweak is initialized using an user-supplied "tweak initialization value" (i), which XEX then encrypts using a second key dubbed "Key 2". As in the case of CTR, the tweak must have the same length as the block cipher's *blocklength.* For each mesage block (the original message is split according to the usual technique as in ECB and CTR), the following block routine is executed:

$$\begin{aligned} T &\leftarrow \mathbb{E}_{key_2}(i) \otimes \alpha^j \\ C_j &\leftarrow \mathbb{E}_{key_1}(M_j \oplus T) \oplus T \end{aligned} \tag{2.2}$$

For the j-th plaintext block the XEX encryption routine acts this way: first, the encrypted tweak initializer is considered as an element of the Galois Field $GF(2^{128})$ and is multiplied by α^j, where α is a primitive element of the aforementioned Galois Field. Then, the j-th plaintext block is encrypted using the tweak we updated before for pre-whitening and post-whitening (in other words, we encrypt the value of plaintext xor tweak, then the obtained ciphertext is xored with the tweak again). The key used in this step is called "key 1".

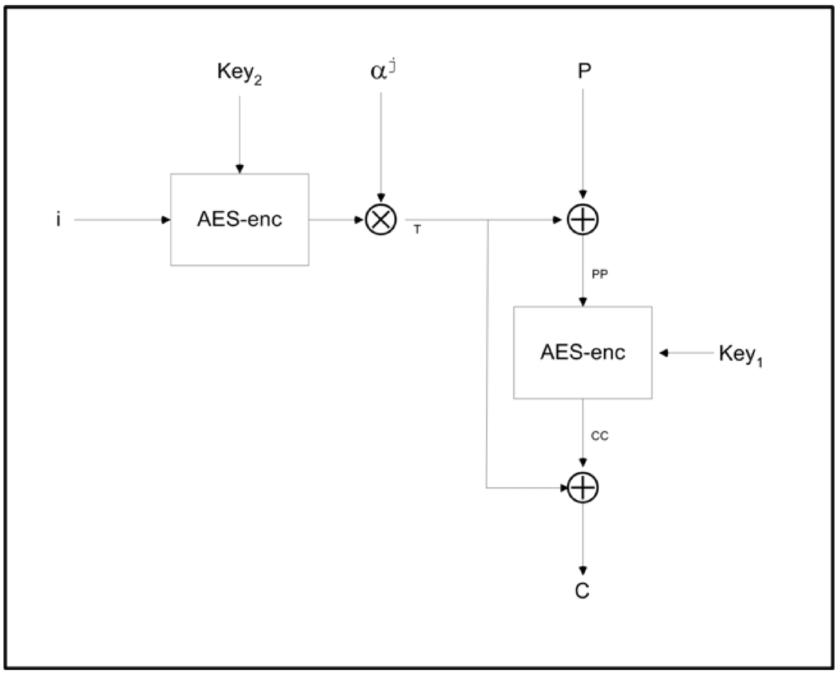

Figure 2.1: Diagram of the XTS-AES block encryption procedure, from [2]

This rather complex block encryption procedure is shown in Figure 2.1; this example makes use of the AES blockcipher. Also note that the XEX block decryption procedure only differs in using decryption with key 1, instead of encryption with key 1:

$$\begin{aligned} T &\leftarrow \mathbb{E}_{key_2}(i) \otimes \alpha^j \\ M_j &\leftarrow \mathbb{D}_{key_1}(C_j \oplus T) \oplus T \end{aligned} \tag{2.3}$$

In our implementation, we used the Galois Field $GF(2^{128})$, with field polynomial $f(x) = x^{128} + x^7 + x^2 + 1$ as per specification [22]. Furthermore, we defined XTS methods for all the block ciphers with a compatible 128 bit block length, even though the specification [2] only covered the use of XTS along with the AES blockcipher.

To get a complete implementation of XTS, we must add a component to perform ciphertext stealing. Ciphertext stealing is a technique used to deal with long messages having a length not multiple of the blocksize; this method can be used alongside any existing mode of operation, but in this paper we will only use it alongside XEX.

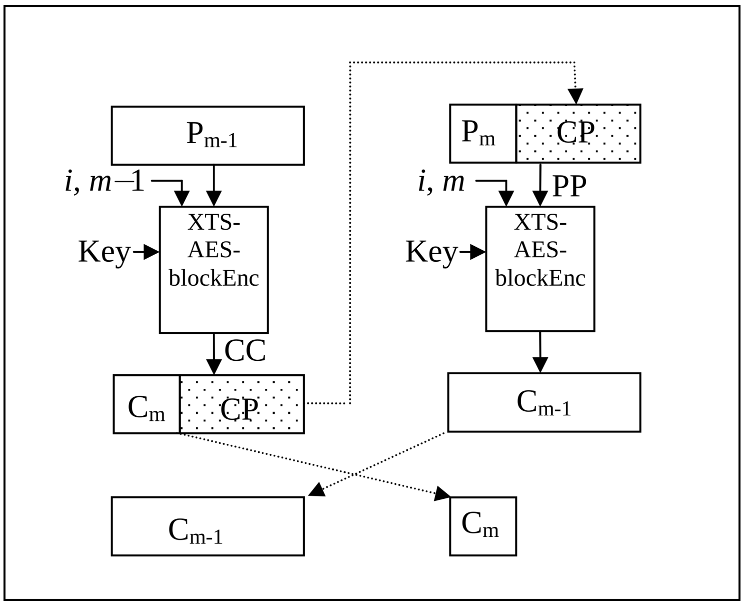

Figure 2.2: XTS-AES encryption of last two blocks when last block is 1 to 127 bits, from [2]

The ciphertext stealing procedure is rather complex, but only involves the last full block plus the last partial block, as shown in Figure 2.2; this diagram for ciphertext stealing uses the XEX block encryption defined above as a procedure.

We assume to be XTS-encrypting a message unevenly split in blocks of size *blocklength*, where the first blocks, numbered from 0 to $m-1$, are precisely *blocklength*-bits long; the last block, with index m, contains a number of bit lower than the *blocklength*. Blocks up to $m-2$ are encrypted using the usual XEX encryption procedure described above; the ciphertext-stealing procedure only involves the last 2 blocks. Ciphertext stealing operates in two steps: first, the last *full* plaintext block (with index $m-1$) is encrypted normally using the XEX procedure; afterwards, the corresponding ciphertext block (called CC in the figure) is not concatenated to the previous ciphertext blocks (with index up to $m-2$); it is instead saved for later re-use. Since the very last plaintext block (with index m) is partial, the missing part is constructed by "stealing" the corresponding bits from the ciphertext CC we saved earlier. The XEX encryption routine is then applied to this "hybrid" block, and the corresponding ciphertext is enqueued to the output array of blocks (in position $m-1$). Lastly, the final partial ciphertext block (index m) is obtained simply by copying the initial bits of CC, the amount of bit copied is equal to the length of the last partial input message block.

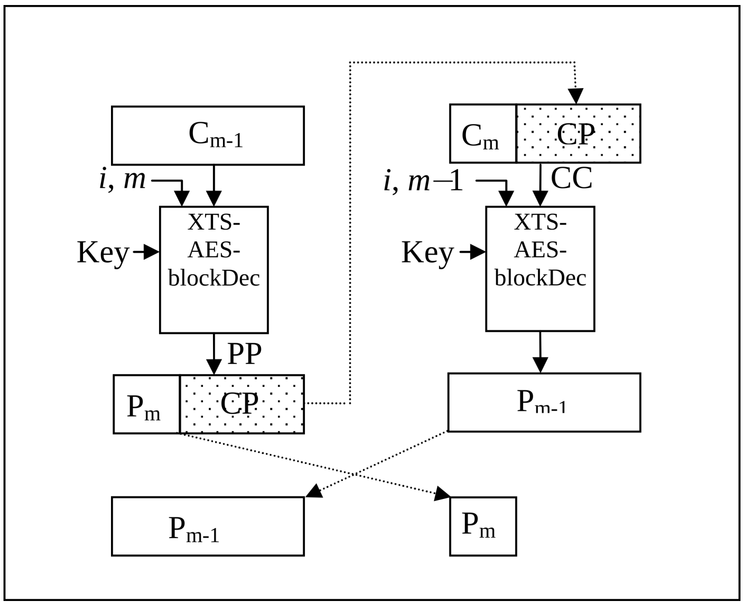

Figure 2.3: XTS-AES decryption of last two blocks when last block is 1 to 127 bits, from [2]

In order to decrypt a message encrypted using XTS, we proceed almost the same way, with two important differencies. First of all, we must obviously call XEX-decryption routines; second, we must swap the tweaks used to compute the last full block and the last partial block, as shown in figure 2.3.

Typically, XTS is used in for encrypting data at rest (i.e. disk encryption); however, XTS does not include any data integrity check of its own, and should be paired with an auxiliary data checksum system to protect data integrity.

Table 2.2: This table lists all the ISO standard ciphers we tested, along with details of their interface and internal structure

Cipher name	**Block length**	**Key size(s)**	**Round structure**
	bit	bit	
AES	128	128, 192, 256	SPN
DES	64	56	Feistel network
Camellia	128	128, 192, 256	Feistel network
CAST5	64	40 up to 128, 8 bit increment	Feistel network
CLEFIA	128	128, 192, 256	Feistel network
HIGHT	64	128	Feistel network
MISTY1	64	128	Feistel network
PRESENT	64	80, 128	SPN
SEED	128	128	Feistel network

Table 2.2 lists all the block ciphers used in the course of our experiments; in the next section, we will describe them one by one, with particular focus on AES.
All real-world block ciphers are designed around a common architecture, divided into two parts: the key schedule and the round structure. The key schedule is a function which expands the user-provided key into a variable number of round keys, used by the round structure. The round structure is a complex routine, composed of a fixed sequence of atomic round functions; the round structure takes the plaintext and the round key as inputs, and yields the ciphertext as output. The round structure also maintains an internal state which is initialized using the plaintext, and is transformed into the ciphertext by the time the encryption is complete. Each atomic round function operates a series of basic transformations on the cipher's internal state, based on the current round number, the current state, and the current round key.

Note that in our project we focused on the round structure design (implemented as an OpenCL kernel); the key schedule was instead run on the host system, using code with little to no modification from the existing reference implementations.

AES [11], originally known as Rijndael, is a widespread block cipher, characterized by an internal structure organized as a Substitution Permutation Network (SPN), a 128 bit block size, and a key length of 128, 192 or 256 bit. The key length determines the number of rounds (respectively 10, 12 and 14 rounds) and the size of the round keys

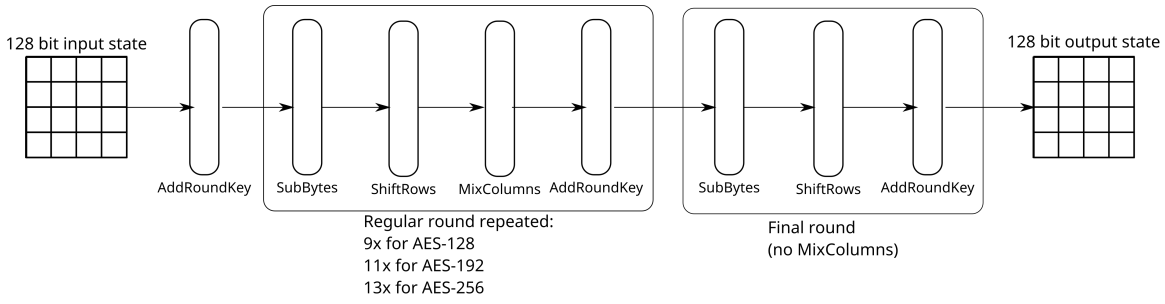

Figure 2.4: AES round structure

expanded in the key schedule (respectively 44, 52 and 60 32-bit words). The single AES round function is constructed as a series of primitive operations (SubBytes, ShiftRows, MixColumns and AddRoundKey) on its internal state; the AES state is a 128-bit number, represented as a grid of 4x4 bytes. At each round, four 32-bit words of round key material is consumed.

Figure 2.4 shows the AES round structure, an example of SPN design: the SPN is built as a repetition of simple round functions which transform their input state into output state. Inside the round function are a series of primitive transformations, each performing a specific task. SubBytes executes substitution of each byte in the 4x4 input state, using a special lookup table (S-box) designed for this purpose. ShiftRows rotates each row in the grid of a fixed amount (respectively, the first, second, third and fourth rows get rotated 0, 1, 2 and 3 places to the left). MixColumns operates on each individual column, which is transformed by multiplication with a fixed 4x4 matrix. Finally, AddRoundKey performs xor addition between the state and the round key.

There are two (plus one) major ways to implement such a round structure: **The "schoolbook" implementation**, which uses 3 separate functions for SubBytes, ShiftRows and MixColumns. A single 256 byte s-box is used inside the SubBytes procedure. For decryption, the round structure must be inverted, and the equivalent functions InvSubBytes, InvShiftRows and InvMixColumns must be invoked. InvSubBytes makes use of a second 256 byte "inverse-s-box". **The "T-tables" implementation** conflates the 3 functions SubBytes, ShiftRows and MixColumns together into 4 lookup tables (256×4 bytes each). For decryption, there are 4 equivalent inverted T-tables, plus one extra 256 byte decryption-only T-table. A **compact variant** of the T-tables method is also available, in which only the first 256×4 byte T-table is stored for both encryption and decryption, and the other 3 tables are generated applying a rotation of 8, 16 or 24 bit to the "master" T-table.

In our study, those 3 styles where dubbed for brevity “small”, “big” and “medium” respectively. As stated in the introduction, we used these AES implementations as a testbed for possible improvements, with the intent of using said improved versions as a basis to develop the other ciphers examined.

Besides AES, only PRESENT is structured as a Substitution Permutation Network; all the other ciphers implement either a textbook Feistel Network or some variations on the basic Feistel formula. PRESENT [9] is a very simple SPN, composed of 31 complete rounds, plus one final key addition. The round function is also very plain, divided in xor-key-addition, substitution using 4-bit S-boxes, and a simple bit permutation. As expected of an SPN cipher, the decryption function simply performs the rounds backwards, and makes use of the reverse substitution and reverse permutation procedures. Among the blockciphers presented here, only DES [1] and SEED [17] are structured as a standard Feistel Network.

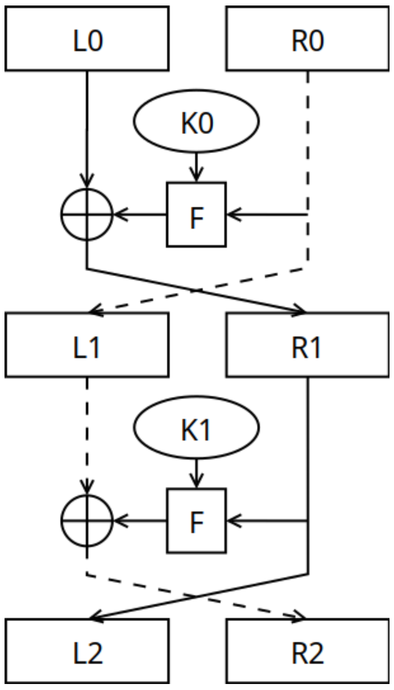

Figure 2.5: Diagram of a simple 2-round Feistel Network

In the “schoolbook” Feistel Network (Figure 2.5), the cipher’s state is made up of 2 half-blocks, initialized splitting the input block in 2 parts. At each round, the “right” half is run trough a Feistel function (F) (which performs key addition, s-box substitution and permutation), and is then xor-added to the “left” half; finally, the two resulting half-blocks are swapped. This scheme allows to re-use the exact same code/hardware for both encryption and decryption, just by inverting the order of the round key used. In the figure above, the round keys are marked as *K0* and *K1*. In the case of SEED, the Feistel function is also implemented as a tiny Feistel network; for this reason, SEED is often classified as a “nested Feistel network”. The other ciphers listed above employ modified versions of the usual Feistel network. Camellia [7], for instance, performs an additional “special round” every 6 Feistel rounds, in which the state is processed using a special function *FL* (and its inverse). Pre and post whitening steps are also used, respectively before the start and at the end of the Camellia round structure. This special design still allows for re-using the same structure for encryption and decryption, provided that the sub-keys (Feistel round key, “special” round key and whitening key) are properly inverted. The reason for this lies in the usage of *FL*; *FL* is applied to the right half of the state, while its inverse is applied to the left half, thus keeping the whole structure invertible. MISTY1 [21] employs a similar structure; however, in this case the same *FL* is applied to both halves, thus requiring to create a stand-alone decryption routine which uses inverse-*FL* in its special round. Another outstanding characteristic of MISTY1 is the usage of different round functions for odd-numbered and even-numbered rounds. CAST5 [3] also uses multiple Feistel functions (3, run cyclically), but the Feistel structure is the standard one. In the case of CAST5, the need for a special decryption routine emerges from the fact that the 3 different functions must be ordered differently. HIGHT and CLEFIA are the most peculiar ciphers, since their structure, known as “generalized Feistel network” strays further from the original Feistel paradigm. In the case of HIGHT [12], the input is split into 8 sub-blocks (also known as *branches*); those branches are then grouped into pairs. Each pair is ran through the usual routine of “process the right half with *F*, add the result to the left”; however, there are no swaps between branches, the entire sub-block array is rotated to the left. HIGHT is also special for the reason that its round functions does not use any S-box at all, but instead relies on modular addition for the “confusion” component. A component of a block cipher is said to add confusion when it makes the relation between key and ciphertext as complex as possible [19]. The blockcipher CLEFIA [25] also employs a generalized Feistel network, albeit with only 4 branches and a more complex round function (with S-box substitution). For the sake of completeness, we must note that CLEFIA’s key schedule does in fact use a 8-branch generalized Feistel network.

Chapter 3

Experimental Evaluation

In this chapter, we present and discuss the experimental data gathered from the OpenCL implementation of the various ISO standard block ciphers introduced in the previous chapter (AES, DES, Camellia, CAST5, CLEFIA, PRESENT, MISTY1, HIGHT and SEED). In the first part we step through various iterations of the AES blockcipher, which we used as a reference algorithm to study various possible design and programming techniques. The various AES implementation encompasses all the design best-practices described in chapter 2, as well as a few techniques which revealed to be unsuccessful. After this vertical perspective, we show the results obtained from the 8 other block ciphers, implemented from the ground-up using the AES best-practices as a reference.

3.1 Test environment

The tests were conducted on an *ATTILA -PROD* board B.1 (RXCA10X115PF40-IDK00A), based on the Altera Arria 10 GX series. The host system is a server running Gentoo Linux (kernel version 4.4.95), with 32 Intel(R) Xeon(R) E5-2620 v4 CPU cores and 125 GB of RAM. The FPGA is connected to the host system using a PCI-E channel. We used the Intel FPGA toolkit version 16.1; the compiler version string (`aoc -version`) is `Version 16.1.0 Build 196`.
The test routine consists of 100 repetitions of the *ECB encryption* procedure with a random payload. After a battery of 100 repetitions, the payload size is increased by 4MB, then the test is repeated. The maximum payload size is capped at 2GB, but usually the test stops much earlier (around 120 MB), when the OpenCL API returns an out of memory error (due to some implementation bug). In the cases where a blockcipher

allows multiple choices for the key length, we have chosen to use the most common:

- AES-128
- Camellia-128
- single DES (56 bit)
- CAST5-80
- CLEFIA-128
- PRESENT-80

3.2 Metrics employed

There is a wealth of available indicators for the performance of our kernels, organized in various tables throughout this chapter. The simplest one is the *throughput* calculated using the test routine, which gives a good blackbox evaluation for the "quality" of a design iteration. As specified before, our test system drives the tested function with payloads of increasing size; therefore, the throughput indicator ("max throughput" in the table) is in fact and indicator of the best throughput across all the block sizes tried, and comes bundled with the best *payload size* value. In later experiments, we introduced an advanced custom profiler, capable of extracting a series of statistics from the test execution timeline; these evaluations are summarized by the *top Kernel/IO* indicator, which express the ratio between the average execution time of the kernel and the average input+output transfer time. Said value is computed over a time-window around the moment when the best throughput was encountered.

Another pair of straightforward indicators is the *fMax* and the *II* (Initiation Interval), as returned from the Intel FPGA OpenCL compiler. As explained before, fMax indicates the maximum frequency at which the FPGA's circuitry can operate, when running a particular `cl_program`; for the FPGA used in our tests, the maximum allowed frequency is 300MHz. The II describes, for each pipelined loop present in the code, the amount of clocks to wait between consecutive iterations of the loop; the ideal II value is 1 (the new iterations are issued with no delay). Since the typical program contains more than one pipelined loop, our table only shows the worst (highest) II. Both fMax and II give an idea of how many iterations per second we'd be able to obtain, without actually running any benchmark.

The compiler also provides a detailed breakdown of the hardware resources used by the OpenCL program; the percentage of *lookup tables (LUT)* occupied, the percentage of *Flip Flops (FF)* occupied, and the percentage of *Block RAM (RAM)* used. On top of those area resource indicators, later experiments also keep track of the kernel compilation resources, in term of total *compiler run-time* and *maximum resident memory* required by the compile processes.

Another useful metric is the *area-time product*; the value of the area term is obtained from the compiler-generated report, which has a specific field for the device's "Logic utilization"; for the FPGA used in our tests, 100% logic utilization corresponds to an absolute logic utilization of 427200. The time term was instead computed as the time needed to process 1 Megabyte of data (simply the reciprocal of the throughput expressed in MB/s).

3.3 Experimental validation of programming practices

In this section we show how the programming practices exposed in the previous chapter can be used to design an increasingly better AES implementation. The data from the upcoming experiments are gathered in a series of tables, and show a subset of the metrics described in the previous section. During the course of our exposition, various diagrams will also be introduced when needed.

As a side note, these first AES variants only implement kernels for ECB encryption, ECB decryption and CTR, XTS will be introduced later on. However, we are less concerned about the XTS kernel as we are for the ECB one, since the benchmarking procedure only involves the ECB routines.

3.3.1 Initial implementation

Since there are no prior examples to show, this first iteration will provide a reference implementation.

The *small* variant is a pretty straightforward translation of the AES round structure (Figure 2.4) into code; a single main loop over the AES rounds, which makes calls to the various functions implementing key addition, row shifting, column mixing and substitution; the substitution is implemented using a single s-box as specified in the standard. In this version, the AES state is kept as a single private buffer, and all the modifications are carried out strictly in-place.

Medium and Big are implemented in a slightly different, more complex way, having the t-table substitutions hidden behind specific C macros; these macros do however require a distinct input and output state to operate, (each output column is a function of all the 4 input columns), therefore in-place modification is not possible. This in turn required to re-structure the main loop to perform two iterations at once, using also an intermediate temporary state. There are also smaller differences, such as having the internal state comprised of 4 32-bit words, rather than 16 8-bit characters; two unrolled loops have the purpose of copying the internal 32-bit state to and from the 8-bit state passed as a parameter to the `encrypt` function.

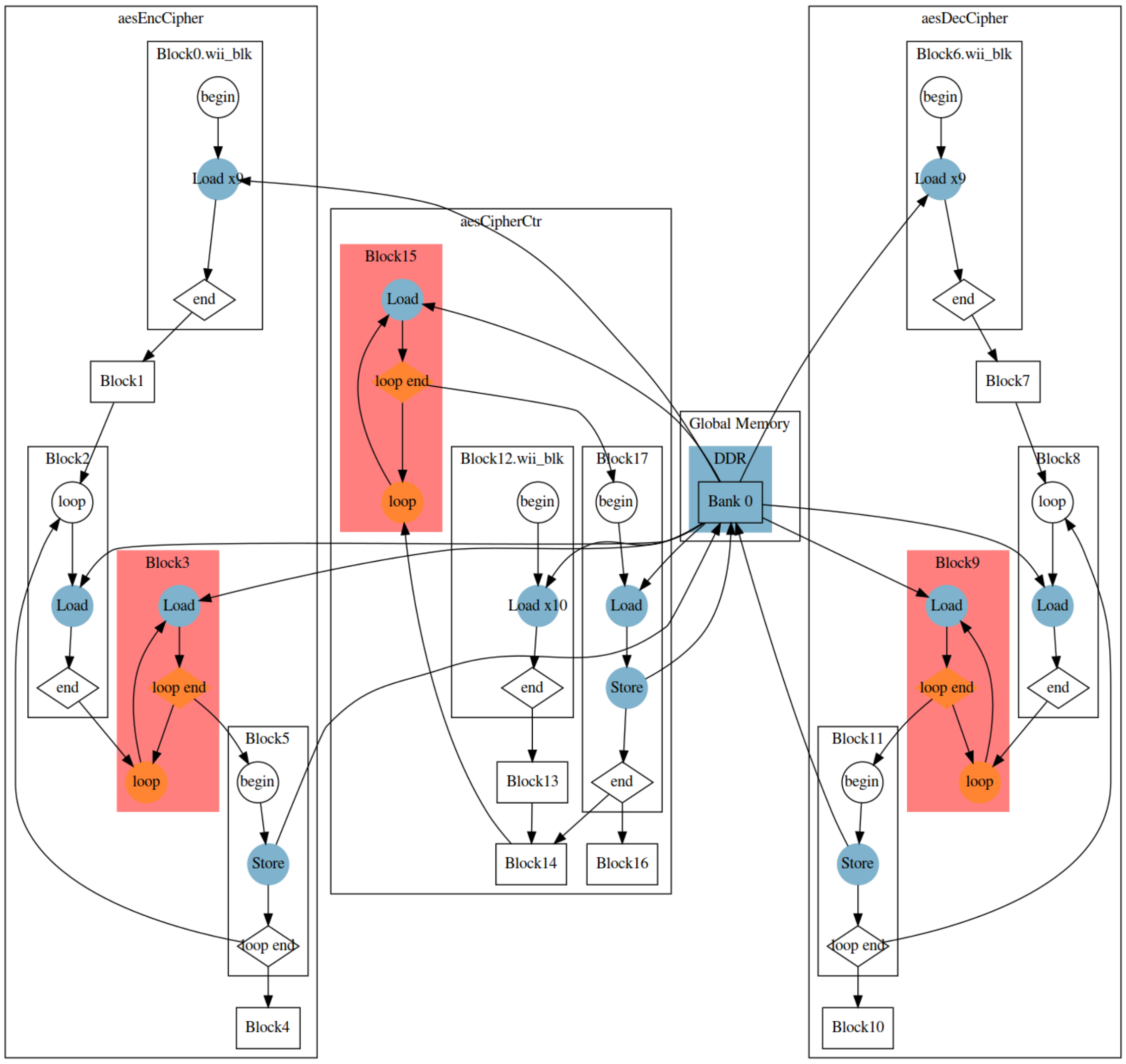

Figure 3.1: aes_medium block diagram, Initial Implementation. Instructions are shown as circles and diamonds, functional blocks are shown as rectangles, memory accesses and control paths are shown as arrows. Global memory and instruction accessing global memory are highlighted in blue; un-optimized function blocks are highlighted in red.

For reference, the compiler-generated block diagram for aes_medium is shown in Figure 3.1. (At this point in the project, medium was used as the standard reference implementation). The diagram shows a simplified logical representation of the aes_medium program, which sits in-between the high level OpenCL code and the low level hardware description language. The diagram is composed of various instructions, represented as circles or diamonds, which are then grouped together into functional blocks (rectangles).

The official Intel report tool offers a web interface, which provides additional comments for each block and instruction, and allows to link each functional block with the corresponding OpenCL lines of code. Both the control flow and the memory accesses are depicted as arrows, whereas the memory resources are represented with colored rectangles. In this graph, the global memory (DDR) is depicted as a blue rectangle; load/store instructions interacting with the global memory are highlighted using the same color. The report tool does also identify badly optimized code blocks and instructions, and highlights them, respectively, in red and yellow, as shown in this example; the specific blocks marked in Figure 3.1 refer to the key-addition functions.

For the sake of brevity, we won't report here the host code used to drive this kernel; it is comprised of a simple sequence of push input, run kernel, get output operations, all running in blocking mode, and is rather similar to the snippet (Listing A.1) provided in chapter 1. This host-side structure will remain pretty much unchanged until later iterations of the project.

3.3.2 Round key cached in local memory

We modified the previous implementation introducing a new logic to store the key locally, and altered the function signatures accordingly. The modification amounted to the addition of a `__local` buffer for storing the key, plus a simple re-implementation of the `memcpy` function.

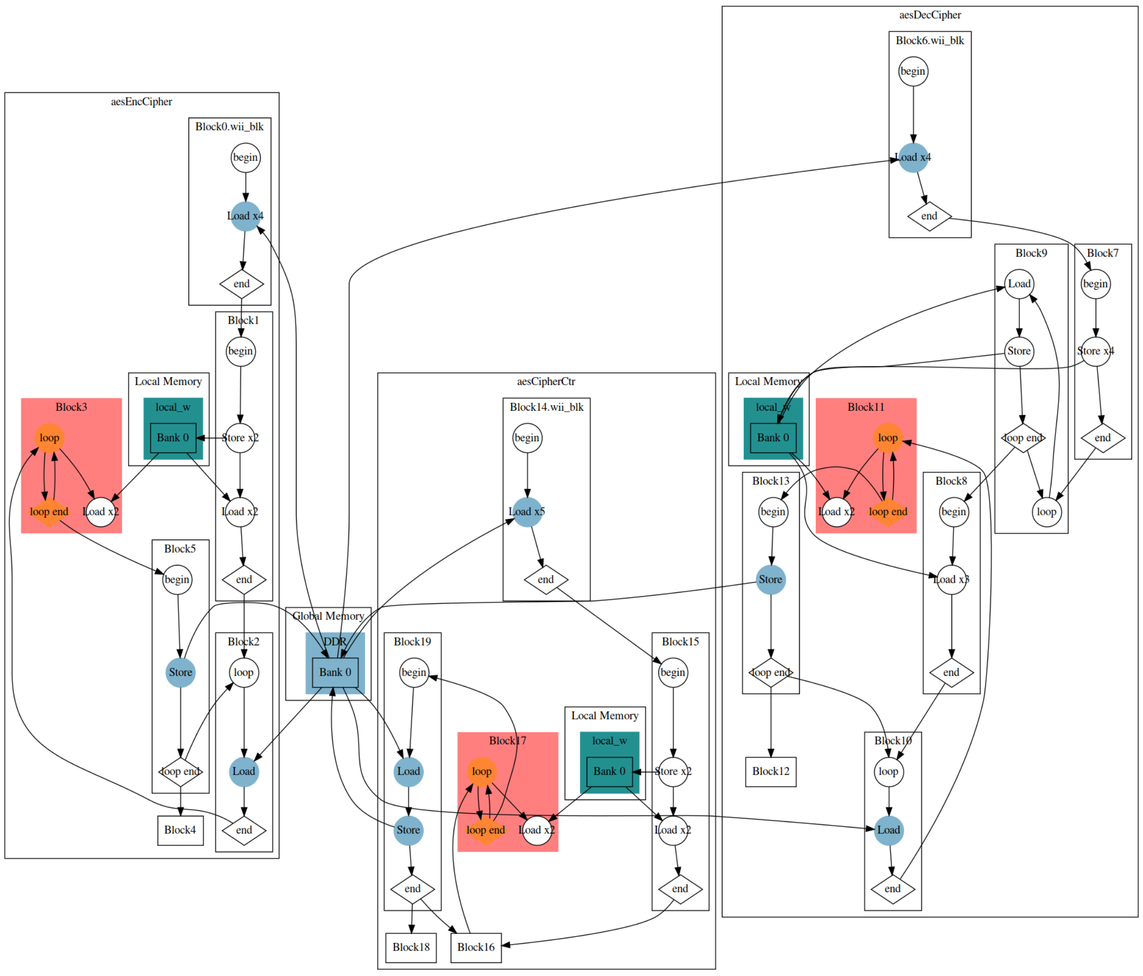

Figure 3.2: aes_medium block diagram, Round key cached in local memory. Instructions are shown as circles and diamonds, functional blocks are shown as rectangles, memory accesses and control paths are shown as arrows. Global memory and instruction accessing global memory are highlighted in blue; local memory blocksare highlighted in green; unoptimized function blocks are highlighted in red.

The block diagram (Figure 3.2) for this iteration of aes_medium shows that indeed a couple of local memory buffers (BRAM) were added. The greenish shade of blue indicates double-pumped local memory banks (double-pumping is a technique which increases the block RAM read speed, at the cost of area).

Table 3.1: Test data gathered from a AES implementations with or without storing the key into local memory, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
Initial implementation	small	13.19	12.08	24.78	-	97
Initial implementation	medium	14.61	13.91	29.55	-	6
Round key cached in local memory	small	17.05	17.2	44.2	207.25	97
Round key cached in local memory	big	14.08	15.24	30.17	244.97	6
Round key cached in local memory	medium	14.1	15.29	30.17	211.86	6

Table 3.2: Test data gathered from a AES implementations with or without storing the key into local memory, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
Initial implementation	small	-	-	-
Initial implementation	medium	333.71	120.0	-
Round key cached in local memory	small	3.60	80.0	29,189
Round key cached in local memory	big	451.82	80.0	212
Round key cached in local memory	medium	411.82	76.0	233

Tables 3.1 and 3.2 show the experimental data gathered from running the various AES variants, with and without using the local memory optimization. This modification resulted in a noticeable improvement in aes_medium's throughput (from 333.71 to 411.82 MB/s, equivalent to a 23% speedup). On the other hand, the performance of kernel "small" was comparatively very poor (3.6 MB/s).

3.3.3 Disable buffer interleaving

Table 3.3: Test data gathered from a AES implementations with or without enabling memory interleaving, part 1 of 2

Technique	Codename	LUT	FF	RAM	fMax	worst II
		%	%	%	MHz	
Interleaving enabled (default)	small	17.05	17.2	44.2	207.25	97
Interleaving enabled (default)	big	14.08	15.24	30.17	244.97	6
Interleaving enabled (default)	medium	14.1	15.29	30.17	211.86	6
Interleaving disabled	small	17.05	17.2	44.2	-	97
Interleaving disabled	big	14.08	15.24	30.17	-	6
Interleaving disabled	medium	14.1	15.29	30.17	-	6

Table 3.4: Test data gathered from a AES implementations with or without enabling memory interleaving, part 2 of 2

Technique	Codename	max	at payload	area-time
	Codename	throughput	size	product
		MB/s	MB	Logic*s
Interleaving enabled (default)	small	3.60	80.0	29,189
Interleaving enabled (default)	big	451.82	80.0	212
Interleaving enabled (default)	medium	411.82	76.0	233
Interleaving disabled	small	3.60	80.0	-
Interleaving disabled	big	452.12	76.0	-
Interleaving disabled	medium	411.71	72.0	-

Compiling the code with memory interleaving disabled (`-no-interleaving default`) resulted in no visible changes in the throughput or the area usage (as shown in tables 3.3 and 3.4). We decided to leave interleaving disabled anyway, as having input, output and round key stored as contiguous buffers made more sense than allowing the compiler to interleave them.

3.3.4 Unified main loop (without unrolling)

Building upon the kernel optimized storing the round key in local memory, we modified all the inner AES routines to accept an input and output state, and introduced temporary buffers as needed. This also allowed us to unify the round structure across the small and medium/big variants: the new loop body performs first the key independent round functions, generating a temporary state; afterwards, the key addition onto the temporary state yields the next actual state. This experiment will serve as a baseline to gauge various loop unrolling techniques.

Table 3.5: Test data gathered from an AES implementation with differentiated main loops (small vs medium/big) and a variant with unified main loops (no unrolling), part 1 of 2

Technique	Codename	LUT	FF	RAM	fMax	worst II
		%	%	%	MHz	
Differentiated main loop	small	17.05	17.2	44.2	207.25	97
Differentiated main loop	big	14.08	15.24	30.17	244.97	6
Differentiated main loop	medium	14.1	15.29	30.17	211.86	6
Unified main loop	small	13.99	12.67	34.35	238.60	10
Unified main loop	big	13.94	13.29	32.65	217.95	2
Unified main loop	medium	13.92	13.33	32.65	237.47	2

Table 3.6: Test data gathered from an AES implementation with differentiated main loops (small vs medium/big) and a variant with unified main loops (no unrolling), part 2 of 2

Technique	Codename	max	at payload	area-time
	Codename	throughput	size	product
		MB/s	MB	Logic*s
Differentiated main loop	small	3.60	80.0	29,189
Differentiated main loop	big	451.82	80.0	212
Differentiated main loop	medium	411.82	76.0	233
Unified main loop	small	269.38	80.0	326
Unified main loop	big	269.54	76.0	327
Unified main loop	medium	289.47	80.0	307

The numeric results were surprising (tables 3.5 and 3.6), as the throughput values of the 3 variants were “leveled” to pretty much the same value: small (269.38 MB/s) saw a huge speed-up (748×), while the performance of medium and big (respectively 289.47 and 269.54 MB/s) was halved.

3.3.5 Main loop full unrolling

Keeping the unified round structure defined before, we tried to improve the main loop at its core, performing loop unrolling in a smart way: the `INNER_AES_LOOP` macro unrolls the main loop by considering that the number of rounds can only be 10, 12 or 14. The code itself isn’t particularly brilliant, just a series of `if` statements used to select the appropriate loop statement; the loop statements have been appropriately decorated using the `#pragma unroll` directive.

Table 3.7: Test data gathered from AES implementations with and without full main loop unrolling, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
No unrolling	small	13.99	12.67	34.35	238.60	10
No unrolling	big	13.94	13.29	32.65	217.95	2
No unrolling	medium	13.92	13.33	32.65	237.47	2
Full unrolling	small	61.46	66.28	171.89	115.67	69
Full unrolling	big	29.57	34.1	121.56	200.24	2
Full unrolling	medium	29.57	34.1	121.56	203.83	2

Table 3.8: Test data gathered from AES implementations with and without full main loop unrolling, part 2 of 2

Technique	Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
No unrolling	small	269.38	80.0	326
No unrolling	big	269.54	76.0	327
No unrolling	medium	289.47	80.0	307
Full unrolling	small	38.37	116.0	9,219
Full unrolling	big	788.30	56.0	316
Full unrolling	medium	789.86	40.0	315

Again, the tuning data (tables 3.7 and 3.8) paints an interesting picture: variants medium (789.86 MB/s) and big (788.3 MB/s) benefited greatly from this new mainloop (2.67× and 2.92× respectively), but small's performance is back to low values (38.37 MB/s). This is easily explained by looking at the resource usage: unrolling the loop made the kernels much more resource-consuming, especially in the case of "small" (61.46% LUT, 66.28% FF, 171.89% BRAM). The block RAM over-use is especially bad; in order to get it back to admissible values, the compiler is forced to reduce (or thoroughly disable) local memory replication; this in turns results in a reduced the fMax and increased pipeline initiation interval.

3.3.6 Manual partial unrolling

A different mainloop was devised: instead of relying on automatic unrolling, the loop underwent a "manual partial unrolling of factor 2"; the number of rounds was halved, and the internal loop body was doubled.

Table 3.9: Test data gathered from AES implementations with different types of loop unrolling, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
No unrolling	small	13.99	12.67	34.35	238.60	10
No unrolling	big	13.94	13.29	32.65	217.95	2
No unrolling	medium	13.92	13.33	32.65	237.47	2
Full unrolling	small	61.46	66.28	171.89	115.67	69
Full unrolling	big	29.57	34.1	121.56	200.24	2
Full unrolling	medium	29.57	34.1	121.56	203.83	2
Manual partial unrolling	small	18.84	21.9	41.02	224.16	134
Manual partial unrolling	big	24.88	29.03	86.47	192.34	1
Manual partial unrolling	medium	24.91	28.95	86.47	202.34	1

Table 3.10: Test data gathered from AES implementations with different types of loop unrolling, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
No unrolling	small	269.38	80.0	326
No unrolling	big	269.54	76.0	327
No unrolling	medium	289.47	80.0	307
Full unrolling	small	38.37	116.0	9,219
Full unrolling	big	788.30	56.0	316
Full unrolling	medium	789.86	40.0	315
Manual partial unrolling	small	430.40	92.0	290
Manual partial unrolling	big	911.90	76.0	216
Manual partial unrolling	medium	923.68	64.0	214

As clearly shown in the data (tables 3.9 and 3.10), the kernels employing this “fat” mainloop were the best performing so far, the resource usage was also reduced from the fully-unrolled variant; however, a significative disparity between the variants small (430.4 MB/s) and medium/big (923.86 and 911.9 MB/s) still remained. Compared to the experiment with no loop unrolling, the speedup of small, medium and big are respectively 1.59×, 3.19× and 3.38×.

3.3.7 Round key forced into local memory, multiple unrolling techniques

This iteration introduced two modifications at the same time: First, we made new a attempt to force the compiler to store the key into registers (normally used for private memory), using the special attribute `register`. Second, we designed new mainloop which combines normal compiler-driven loop unrolling with the "fat" loop body defined earlier. This version is also special in having a single unrolled loop rather than three distinct ones; `if` statements are used to append "extra rounds" after the main loop.

Table 3.11: Test data gathered from this AES implementation, compared with the no-unrolling baseline, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
No unrolling	small	13.99	12.67	34.35	238.60	10
No unrolling	big	13.94	13.29	32.65	217.95	2
No unrolling	medium	13.92	13.33	32.65	237.47	2
Combined unrolling, key in registers	small	18.55	16.46	61.65	267.52	1
Combined unrolling, key in registers	big	18.26	17.93	69.83	238.37	15
Combined unrolling, key in registers	medium	18.23	17.85	69.6	255.55	15

Table 3.12: Test data gathered from this AES implementation, compared with the no-unrolling baseline, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
No unrolling	small	269.38	80.0	326
No unrolling	big	269.54	76.0	327
No unrolling	medium	289.47	80.0	307
Combined unrolling, key in registers	small	961.25	60.0	110
Combined unrolling, key in registers	big	936.95	44.0	168
Combined unrolling, key in registers	medium	946.65	72.0	166

All the 3 implementations managed to reach a high throughput using this these new modifications (the data are shown in tables 3.11 and 3.12); more surprisingly, aes_small is now the fastest in the bunch (961.25 MB/s, against the 946.65, 936.95 MB/s of medium, big). The speedups of small, medium and big are respectively 3.57×, 3.48× and 3.27× from the no-unrolling baseline. The explanation is that, now that the round key is stored within private memory, the reads from lookup tables in *constant* memory become the new bottleneck since, on this FPGA model, the constant memory is implemented using a fixed size cache in block RAM.

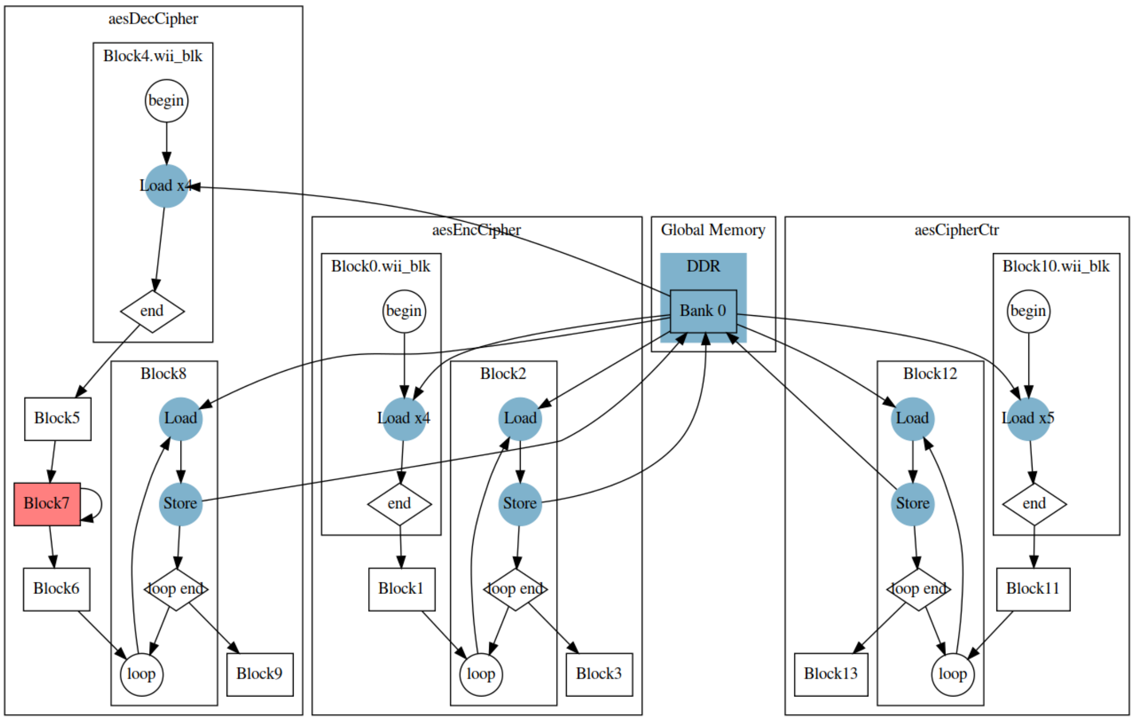

Figure 3.3: aes_medium block diagram for the variant with round key forced into private memory and combined unrolling techniques. Instructions are shown as circles and diamonds, functional blocks are shown as rectangles, memory accesses and control paths are shown as arrows. Global memory and instruction accessing global memory are highlighted in blue.

The block diagram resulting from this AES version (Figure 3.3) shows how moving the round key into private memory resulted in a more streamlined implementation, with much less load/store operations.

Given that this is the best result reached in a non-multi-buffered kernel, this AES iteration (and the small variant in particular) will be elected as the baseline for the following implementations.

3.3.8 Extracting **`num_rounds`** as a constant

The next logical step was to extract the control-flow dependency on the value of `num_rounds` (the parameter used to store the number of AES round to perform), by means of creating 3 copies of each kernel with the 3 different `num_rounds` set as compile-time constants.

Table 3.13: Test data gathered from AES implementations with a single entry point and AES implementations with multiple entry point (3 in total), part 1 of 2

Technique	Codename	LUT	FF	RAM	fMax	worst II
		%	%	%	MHz	
Single entry point	small	18.55	16.46	61.65	267.52	1
Single entry point	big	18.26	17.93	69.83	238.37	15
Single entry point	medium	18.23	17.85	69.6	255.55	15
Multiple entry points	small	29.97	32.14	152.00	221.04	1
Multiple entry points	big	31.31	36.15	168.05	215.42	15
Multiple entry points	medium	31.31	36.16	168.05	217.86	15

Table 3.14: Test data gathered from AES implementations with a single entry point and AES implementations with multiple entry point (3 in total), part 2 of 2

Technique	Codename	max	at payload	area-time
	Codename	throughput	size	product
		MB/s	MB	Logic*s
Single entry point	small	961.25	60.0	110
Single entry point	big	936.95	44.0	168
Single entry point	medium	946.65	72.0	166
Multiple entry points	small	910.69	68.0	212
Multiple entry points	big	911.37	44.0	358
Multiple entry points	medium	911.92	56.0	358

In practice, this approach resulted in a slight decrease in the overall performance (data shown in tables 3.13 and 3.14), and was therefore abandoned.

3.3.9 Interlude 1: Strategies to parallelize over multiple work items

Having reached a satisfactory performance for a pure single-work-item kernel, the next logical step was attempting to scale the architecture horizontally, by replicating the single-work-item kernel multiple times in order to use all the available resource, and benefit from the speedup given by 2 or even 4 pipelines working in parallel.

It is worth mentioning that at this point we did also explore a data-parallel multi-work-item implementation, but it performed poorly both resource-wise and performance-wise, and was therefore abandoned quickly.

In order to achieve effective task parallelism, 2 major strategies were devised. Strategy 1: Make use of the *intel_ channels* extension (originally called *altera_ channels*), which allows the user to create inter-kernel FIFO objects, in order to implement both the multi-pipeline and its control logic entirely on device-side.

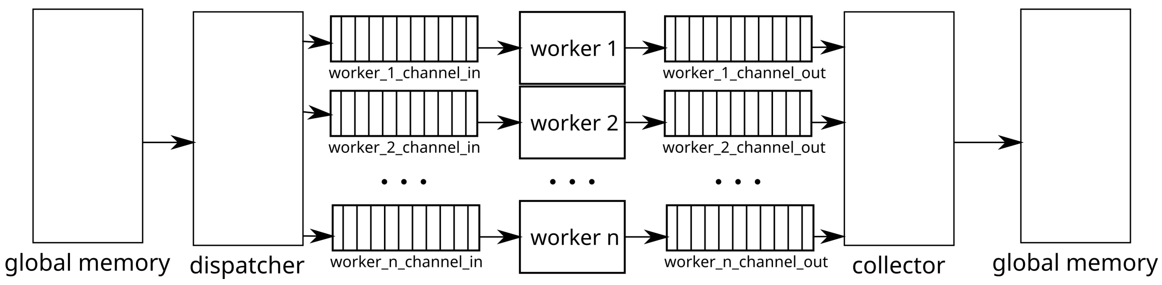

Figure 3.4: Multi-pipelined kernel architecture using Intel channels

The architecture, shown in Figure 3.4 is comprised of a "dispatcher" kernel, which reads data blocks from global memory and forwards them to the appropriate input channel; each of these channels is polled by a "worker" kernel, which performs the required encryption and enqueues the data into its own output queue. Finally, a "collector" kernel moves the output from the channels to the global memory buffer. Strategy 2: Create multiple aliases of each kernel and manage them independently via host logic, using multiple `cl_kernel` objects, `cl_mem` buffers and `cl_command_queue` as needed.

Table 3.15: Test data gathered from the baseline single work-item AES implementation and the channel-based multi work-item implementation outlined in strategy 1, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
Single work-item baseline	medium	18.23	17.85	69.6	255.55	15
Channel-based multiple work-item implementation	medium	33.03	34.93	131.02	224.41	15

Table 3.16: Test data gathered from the baseline single work-item AES implementation and the channel-based multi work-item implementation outlined in strategy 1, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
Single work item baseline	medium	946.65	72.0	166
Channel-based multiple work-item implementation	medium	876.58	36	343

Despite being the most promising, strategy 1 didn't perform well, since reading/writing to the channels introduced an extra latency we couldn't manage to get rid of. The data from our experimentation with channels (tables 3.15 and 3.16) show a throughput of 876.58 MB/s, worse than the ones obtained previously using the pure single-work-item kernel. With that considerations, we resolved to proceed forward developing strategy 2.

As a sidenote, strategy 2 was tested using compile units containing only ECB encrypt kernels, in order to devote all available resources to replicate the specific kernel used for benchmarking.

Up until this moment, we made use of the official Intel report tool and profiler. However, during the developement of AES variants derived from strategy 2, the shortcomings of the official profiler became evident, and we resorted to create a custom profiler of our own, which overlays an in-depth timeline of the kernel lifetime with useful statistics computed over a sliding window.

3.3.10 2 separate workers

We implemented multiple workers leveraging the task-parallelism available in OpenCL. On the OpenCL code side, it was simply a matter of creating multiple copies of the same kernel, using an appropriate naming scheme. We achieved this exploiting the metaprogramming capabilities of C by means of a `DECLARE_WORKER_ENC` macro, which is expanded into an OpenCL kernel with a certain naming scheme. The declare-worker macro is then supplied to another macro, called `FOREACH_WORKER()`. The for-each macro is expanded by applying the argument macro (in this case, the declare-worker macro), to the list $\{0, 1\}$. In this case, the foreach macro is used to declare multiple identical worker kernels, with names "aesEncCipher_0" and "aesEncCipher_1".

The host code was entirely restructured according to the algorithm described in Algorithm 2.3.1, with the significant difference that the input chunk is split evenly across the workers, rather than having the workers process separate chunks altogether.

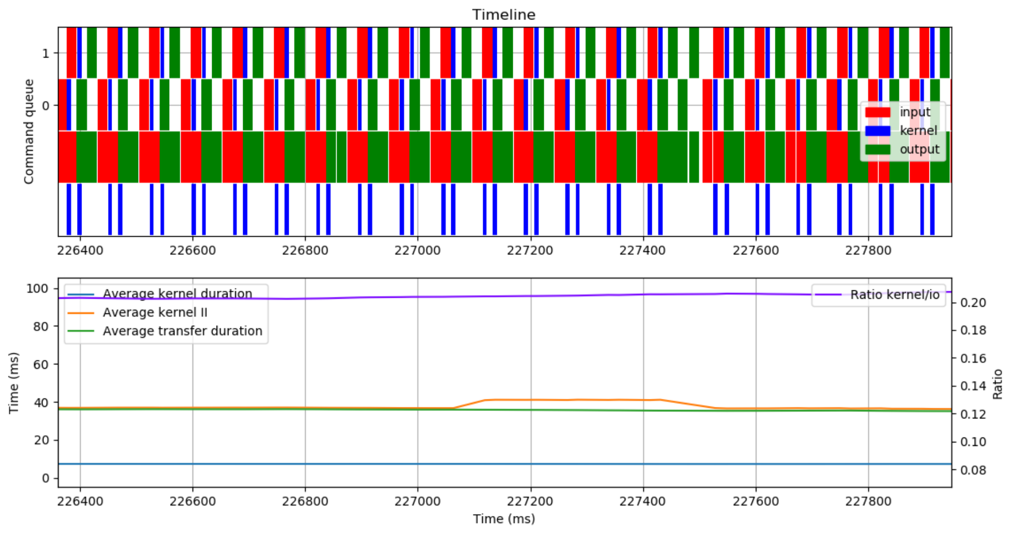

Figure 3.5: Timeline diagram for aes_small using 2 workers, zoomed over a section of the test schedule

Figure 3.5 show a fragment of the tuning routine, taken using our custom profiler.

The custom profiler is implemented as a modular component of the host OpenCL application; it is based around a series of OpenCL API which allow to precisely time OpenCL events. First, the command queue(s) employed must be initialized with the proper flag (`CL_QUEUE_PROFILING_ENABLE`); second, whenever the user wishes to profile an enqueued operation, the corresponding `clEnqueue*` function must be set to return a `cl_event` object, as shown in the previous chapter 2.3.2. The `cl_event` returned can be used to profile the enqueued operation: to do so, the user must call the `clSetEventCallback()` API, which allows to attach an user-supplied callback to an OpenCL event; the callback is run when the event reaches the "complete" state. As per OpenCL specification, the user-defined callback receives the triggering `cl_event` as a parameter: we implemented the body of the callback as to invoke the `clGetEventProfilingInfo()` function on that particular event. This last API call allows the user to obtain various timestamps related to the life-cycle of the operation linked to a particular event; the time is returned in nanoseconds, computed using the device's own time counter. In our case, we used it to retrieve the start and the end of the execution, but other settings also allow to retrieve the time at which the operation is enqueued and submitted to the device.

The resulting diagram is divided in two sub-graphs. The upper diagram shows a high-level timeline of the device operation, displaying input transfers, kernel executions and output transfers in different colors. The operations are divided into different lanes, depending on the `cl_command_queue` on which they were enqueued; the two anonymous lanes at the bottom summarize all the transfers and all the executions across all command queues. The lower diagram shows, on the same time-scale, a series of statistics on the operations shown above, computed using a sliding window approach. "Average kernel duration" and "Average transfer duration" are pretty straightforward and self-explanatory; "Average kernel II" measures the time interval in-between consecutive "kernel execution start" events. Finally, "Ratio kernel/IO" indicates the ratio between the value of "Average kernel duration" and "Average transfer duration". Notice the double "output read" between 227400 and 227600 milliseconds; this is not an error, but rather a side effect of the re-synchronization phase, which was implemented as a "blocking read".

Table 3.17: Test data gathered from the baseline single work-item AES implementation and the multi work-item variant using 2 workers, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
Single work-item baseline	small	18.55	16.46	61.65	267.52	1
Multiple work-item implementation (2 workers)	small	21.57	20.11	80.42	249.25	1

Table 3.18: Test data gathered from the baseline single work-item AES implementation and the multi work-item variant using 2 workers, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
Single work item baseline	small	961.25	60.0	110
Multiple work-item implementation (2 workers)	small	1,458.67	108	87

Performance-wise, the benchmark did well (1458.67 MB/s for the small implementation, with a speedup of 1.51× from the baseline single work-item version), but did not quite double the baseline throughput (shown in tables 3.17 and 3.18), as we wished it would.

3.3.11 4 separate workers

With minimal modifications to the structure of host and kernel code, we were able to adapt our infrastructure to handle 4 workers at a time.

Table 3.19: Test data gathered from the baseline single work-item AES implementation and the multi work-item variant using 2 workers, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
Single work-item baseline	small	18.55	16.46	61.65	267.52	1
2-workers implementation	small	21.57	20.11	80.42	249.25	1
4-workers implementation	small	21.57	20.11	80.42	249.25	1

Table 3.20: Test data gathered from the baseline single work-item AES implementation and the multi work-item variant using 2 workers, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
Single work item baseline	small	961.25	60.0	110
2-workers implementation	small	1,458.67	108	87
4-workers implementation	small	1,398.58	68	91

Tables 3.19 and 3.20 show a comparison between the test data of this 4-workers implementation, the 2-workers implementation and the single work item baseline. Surprisingly, the performance of this AES variant (1398.58 MB/s) was even worse than that of the 2-workers version.

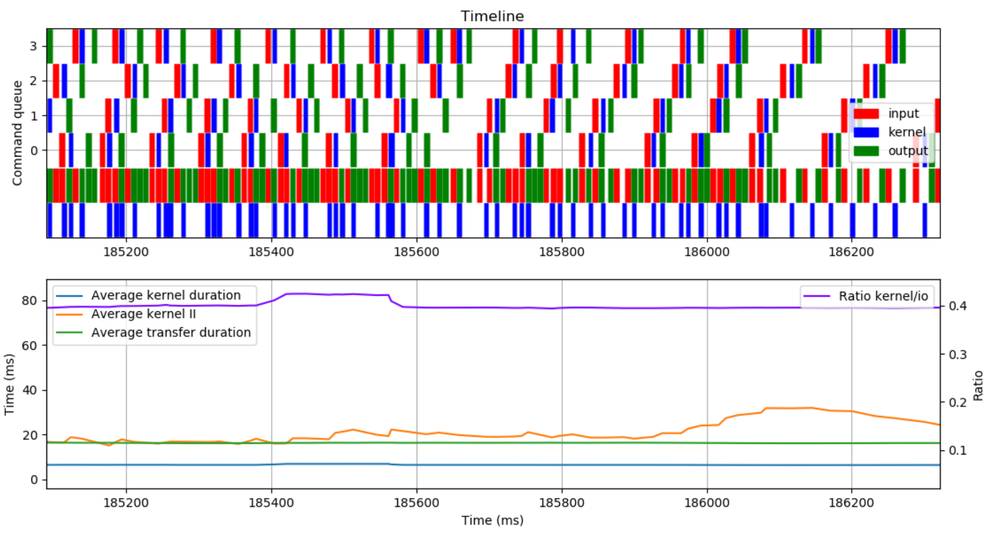

Figure 3.6: Timeline diagram for aes_small using 4 workers, zoomed over a section of the test schedule

As shown in the timeline diagram (Figure 3.6), the execution profile changes dramatically. Since we didn't actually impose a restrictive schedule for the transfers/kernels, OpenCL's own scheduling system took over, resulting in a very chaotic timeline. The execution profile shown can be split easily into two parts (divided by the re-synchronization reads, visible around 18.56s):

1. the left part, with a more chaotic profile, but an overall better performance (higher kernel/io time ratio, lower kernel II)
2. the right part, with a more regular profile, but a clearly worse performance

Keen-eyed readers will also notice how, in a couple of instances, the kernels on the left half also happen to overlap a little. This is significant, since, in the previous experiment, having two distinct workers didn't make the execution profile any different from using a single worker with double-buffering.

3.3.12 Interlude 2: performance wall

From the last series of experiments, we derived 2 considerations:

- The performance benefits from having overlaps of kernel executions

- Lowering the block size causes the transfer time to shrink faster than the kernel execution time

With that in mind, we attempted to modify our infrastructure in order to enqueue smaller block sizes, scheduling the transfers in a way that would cause them to overlap.

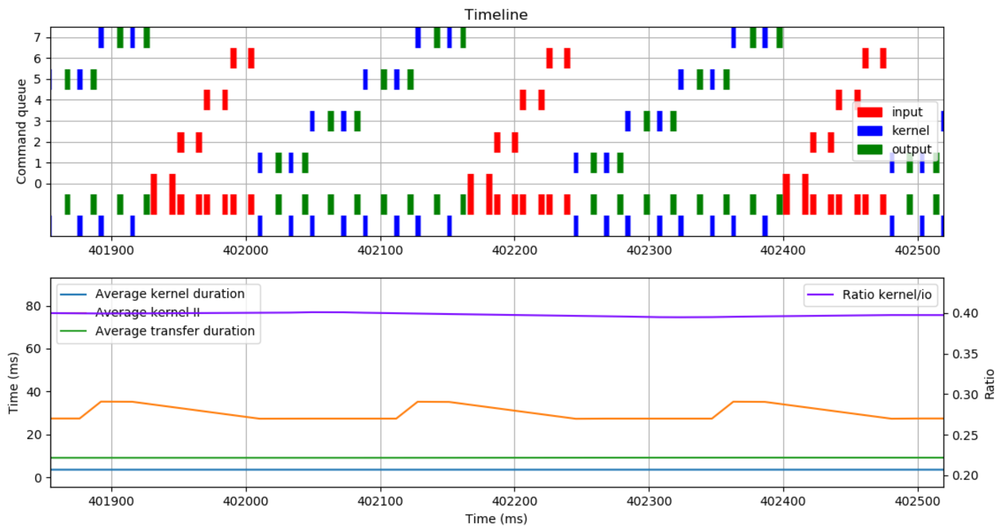

Figure 3.7: Timeline diagram for aes_small using 4 workers and 8 buffers, 1/8 MB blocksize, zoomed over a section of the test schedule

However, we discovered that very small transfers incur in a significant overhead, which in turn decreases the efficiency of the input/output transfer, as shown in Figure 3.7.

We framed the overhead problem as caused by the host side infrastructure, and tried to solve it with a series of improvements in the multi-workers management code. The host side code was indeed made more streamlined and efficient, but the overhead remained pretty much unchanged. In the end, we decided to give up and try a different approach.

3.3.13 NO-OP test

We started over, using the 2-workers variant, the all-time best so far, as a baseline. We devised a sort of "No-Op" cipher, which would only enqueue write and read operations, but won't execute any kernel at all.

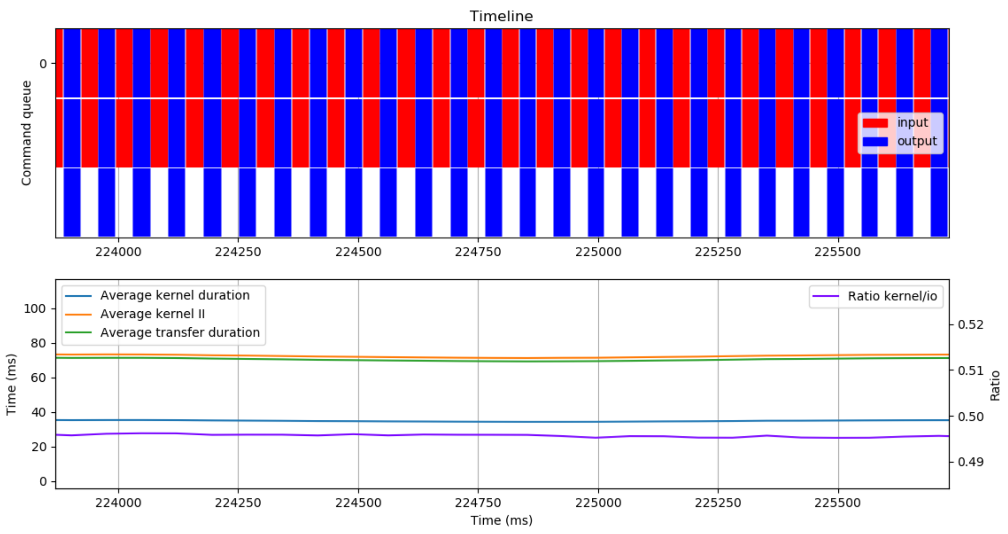

Figure 3.8: Timeline diagram for the "No-Op" cipher

The timeline of "No-Op" is shown in Figure 3.8; despite not executing any kernel, we introduced "dummy" kernel events (bottom unnamed lane), as to avoid glitches in the routines used to generate the bottom diagram (which, needless to say, is meaningless in this specific case).

Table 3.21: Comparison of the throughput of the 2-workers AES variant against a No-Op application

Technique	Codename	max throughput	at payload size
		MB/s	MB
2-workers implementation	small	1,458.67	108
No-Op	No-Op	1,491.90	40

The throughput value (1491.9 MB/s, as shown in Table 3.21) is very close to that of the 2-workers implementation, confirmed our suspects: the device-host channel is the factor limiting our maximum throughput.

3.3.14 Pre-final AES design

Thanks to the insight gathered in No-Op test, we concluded that our best choice for the final design is:

- A single work-item kernel with no replication

- A host-side architecture using double-buffering

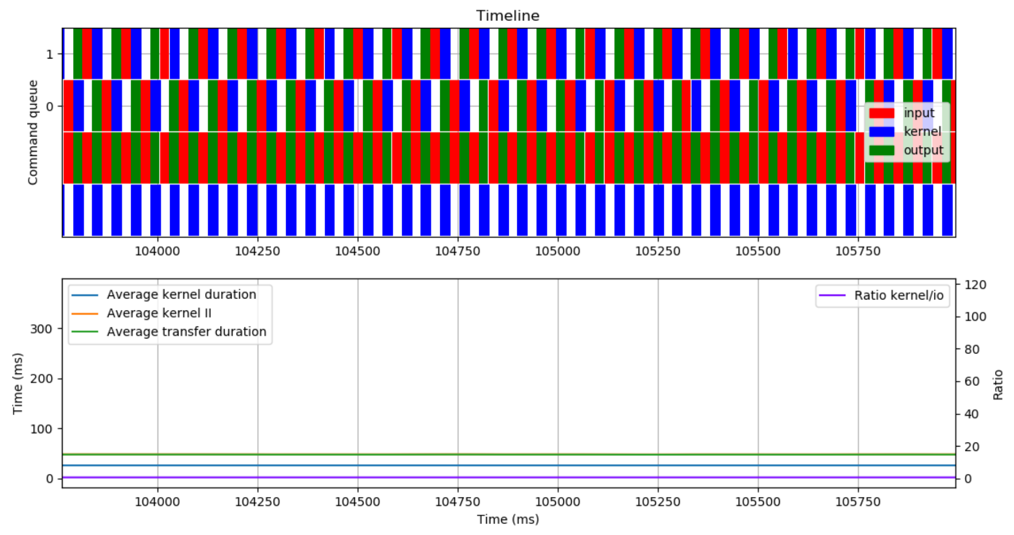

Figure 3.9: Timeline diagram for aes_small single work item with double buffering, zoomed over a section of the test schedule

The final aes_small iteration is shown in action in Figure 3.9.

Table 3.22: Test data gathered from the baseline single work-item AES implementation, two-workers implementation and double-buffered single work-item implementation, part 1 of 2

Technique	Codename	LUT %	FF %	RAM %	fMax MHz	worst II
Single work-item baseline	small	18.55	16.46	61.65	267.52	1
2-workers implementation	small	21.57	20.11	80.42	249.25	1
Single work-item double-buffered	small	46.33	53.79	262.00	180.31	1

Table 3.23: Test data gathered from the baseline single work-item AES implementation, two-workers implementation and double-buffered single work-item implementation, part 2 of 2

Technique	Codename Codename	max throughput MB/s	at payload size MB	area-time product Logic*s
Single work item baseline	small	961.25	60.0	110
2-workers implementation	small	1,458.67	108	87
Single work-item double-buffered	small	1,475.06	76	226

Tables 3.22 and 3.23 compare the results obtained from the baseline pure single work-item implementation, the multiple work-item implementation (2 workers) and this double-buffered single work-item implementation. It is at this point that we introduced the first version of the XTS mode of operation; this first XTS kernel was resource hungry and not fully optimized, resulting in a slightly lower fMax and a higher kernel/io ratio in the profiler diagram. Later AES versions attempt fix this, by both improving the code and moving the XTS kernels to their own, separate OpenCL programs.

3.3.15 Final summary

To give a visible comparison of the throughput of the various AES implementations examined, we present a couple of plots, showing raw data gathered through the test routine. The diagrams show the values assumed by the blockcipher throughput across different payload sizes.

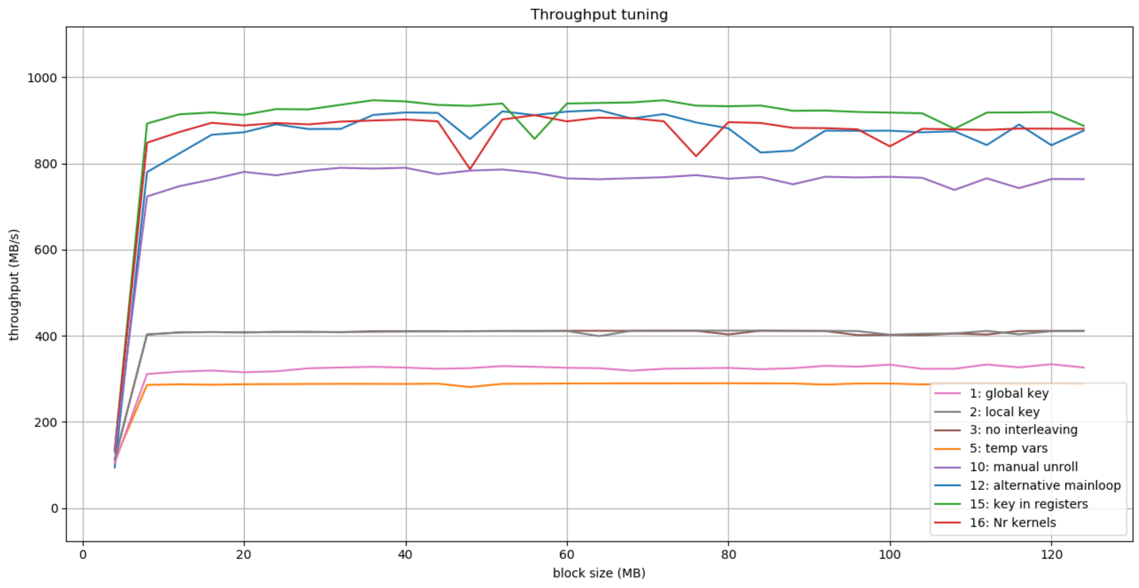

Figure 3.10: Performance of the aes_medium kernel across different design iterations (pure single work item kernel, without any form of multi-buffering)

Diagram 3.10 shows a comparison of the test data gathered from experiments using only single work item kernels.

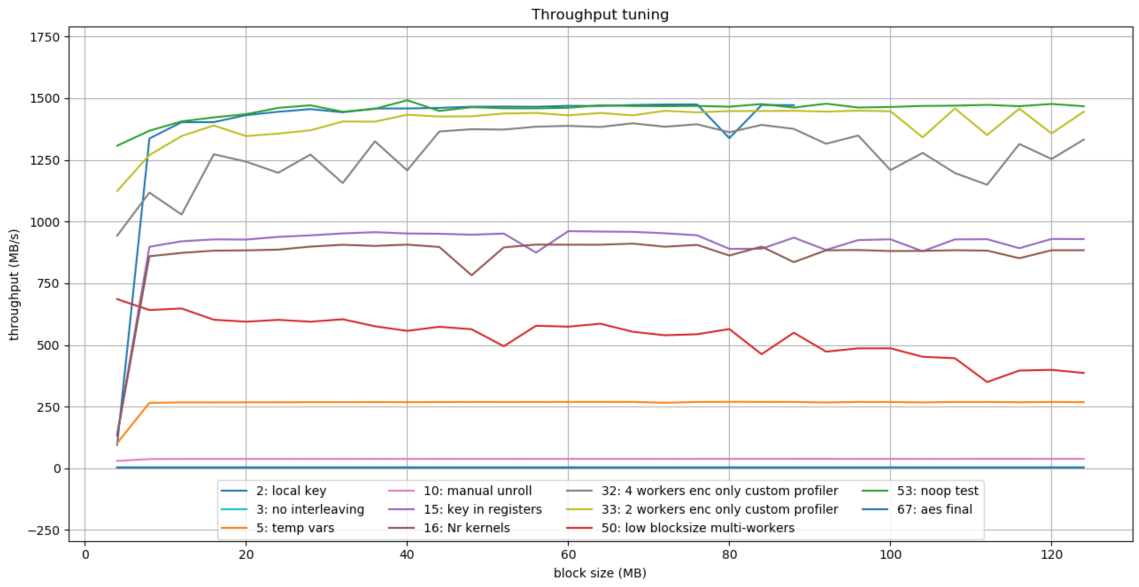

Figure 3.11: Performance of the aes_small kernel across different design iterations

Figure 3.11 summarizes performance data gathered from benchmarks of aes_small, encompassing all the design iterations described so far in this chapter.

3.4 Horizontal slice

The following tables collect raw data obtained from compiling and benchmarking the blockciphers listed in table 2.2; the following implementations make use of all the best practices described above (double-buffered single work-item kernel, local/private memory caching, loops unrolled using a combination of automatic and manual strategies). Note that, for this "big bang test", we used an AES implementation split across 3 OpenCL programs: part 1 includes kernels for ECB encryption, ECB decryption and CTR; part 2 includes a single kernel for XTS encryption; part 3 includes a single kernel for XTS decryption.

Table 3.24: Test data gathered from the big bang test of all the studied block ciphers, part 1 of 2

Cipher name	Cipher variant	LUT %	FF %	RAM %	fMax MHz	Worst II	area-time product
AES	single work item, no XTS	16.88	16.08	61.85	254.84	1	69
AES	single work item, only XTS-encrypt	19	18.85	84.3	179.53	8	81
AES	single work item, only XTS-decrypt	31.47	33.78	133.42	126.45	1	166
Camellia	single work item	53.17	64.35	231.95	200.92	8	204
CAST5	single work item	12.89	10.88	24.54	268.6	1	62
CLEFIA	single work item	43.05	52.56	239.59	177.55	8	203
DES	single work item	19.59	19.44	90.5	231.85	1	58
HIGHT	single work item	11.37	8.55	9.42	189	1	45
MISTY1	single work item	11.69	9.45	28.58	258.06	1	40
PRESENT	single work item	18.54	18.96	124.82	247.21	1	50
SEED	single work item	38.04	45.61	220.98	201.57	8	163

Table 3.25: Test data gathered from the big bang test of all the studied block ciphers, part 2 of 2. For space concerns, "Payload size" has been abbreviated to "PL size" and "Top Kernel/IO" has been abbreviated to "Top K/IO"

Cipher name	Cipher variant	Max throughput MB/s	PL size MB	Top K/IO	Compiler run-time hh:mm:ss	Max resident mem MB
AES	single work item, no XTS	1,496.53	68	0.401	4:24:53	18,762.26
AES	single work item, only XTS-encrypt	-	-	-	8:37:31	19,554.85
AES	single work item, only XTS-decrypt	-	-	-	35:55:05	38,098.16
Camellia	single work item	1,477.66	20	0.503	11:07:59	37,157.75
CAST5	single work item	1,496.75	60	0.713	3:15:15	15,096.59
CLEFIA	single work item	1,499.38	60	0.565	10:25:19	36,416.80
DES	single work item	1,480.57	24	0.861	3:34:23	15,764.96
HIGHT	single work item	1,360.65	56	1.025	3:01:06	13,368.93
MISTY1	single work item	1,503.68	24	0.785	2:41:00	13,452.62
PRESENT	single work item	1,444.56	12	0.796	3:21:59	14,435.27
SEED	single work item	1,493.09	16	0.494	8:22:35	31,630.55

There are some interesting observations to be made, based on the raw data obtained in the "big bang test" (tables 3.24 and 3.25). First, we can classify the blockciphers into 2 broad sets of "expensive" and "cheap" ciphers, depending on the amount of resource consumed (both in terms of area and compile-time/memory). It's immediately evident, when also comparing the results with table 2.2, that only the ciphers with a 128-bit block belong to the "expensive" group. This is simply due to the fact that having a 128-bit block size allows us to also define kernels for the XTS mode of operation [2]; the 64-bit block ciphers instead did not include XTS kernels, as the specification only described XTS over a 128-bit block size. That being said, the test procedure only encompasses the ECB encryption procedure; the XTS routines are implemented but not benchmarked. Regarding the XTS implementation in OpenCL, it really pushes the limit of the current Intel OpenCL compiler; for this reason, we developed it as a trade-off in efficiency and complexity. This shows well in the split aes data: despite being almost identical codewise, the program for XTS encryption uses much less resources (and is much slower) than its decryption counterpart; also note how the XTS encryption kernel is the one responsible for all the "bad" Initiation Intervals equal to 8. This bad II doesn't actually appear in the throughput measurements, as the test routine only invokes the ECB encryption primitive. 128-bit block ciphers other than AES were kept as a single compile unit for the sake of simplicity, but, as shown by the performance reports, their throughput did not suffer from this choice. Another surprising set of results is given by HIGHT: despite having a very low resource usage, its fMax and throughput are the lowest in the series; indeed, the Kernel/IO ratio is also very large, at the point that the kernel runtime, rather than the input/output transfer, becomes the bottleneck. We suspect that the cause for this slowness lies in the design of HIGHT, which relies heavily on modular addition operations. Being devoid of s-boxes, HIGHT gains its non-linearity from the carry paths in addition operations, however, these same carry lines also constitute critical paths inside the logic circuit, slowing down the whole system. It would be interesting, for future research, to test a HIGHT implementation structured as a data-parallel NDrange kernel, rather than a task-parallel single work item kernel as used here. With the exception of HIGHT, all the ciphers in exam reach the maximum throughput value allowed by the limited device-host channel.

Table 3.26: Speculative values computed from the results in 3.24 and 3.25

Cipher name	Cipher variant	Throughput unconstrained MB/s	Worst design throughput MB/s	Best design throughput MB/s
AES	single work item, no XTS	3,732.00	3,888.55	3,888.55
AES	single work item, only XTS-encrypt	-	342.43	2,739.41
AES	single work item, only XTS-decrypt	-	1,929.47	1,929.47
Camellia	single work item	2,937.69	383.22	3,065.80
CAST5	single work item	2,099.22	2,049.26	2,049.26
CLEFIA	single work item	2,653.77	338.65	2,709.20
DES	single work item	1,719.59	1,768.88	1,768.88
HIGHT	single work item	1,360.65	1,441.96	1,441.96
MISTY1	single work item	1,915.51	1,968.84	1,968.84
PRESENT	single work item	1,814.77	1,886.06	1,886.06
SEED	single work item	3,022.44	384.46	3,075.71

Table 3.26 shows speculative throughput values, computed directly from the data shown in 3.24 and 3.25. *Throughput unconstrained* by PCI-E transfer indicates the maximum possible throughput, without considering input/output bandwidth limitations; naturally, this indicator only makes sense for kernels having a Kernel/IO (KIO) lower than 1.0. Unconstrained throughput is computed as:

$$X_{unconstrained} = X_{io\_bound}/KIO \tag{3.1}$$

The intent of this indicator is to predict what performance we'd be able to attain on an upgraded system, with a faster transfer channel between main system memory and accelerator board. However, at the time of writing we don't have access to such a system, and therefore we can't evaluate the accuracy of this indicator.

Another interesting value is the *Design throughput.* The value is computed considering the ideal case in which the kernel can operate processing one cryptographic block (see table 2.2) at the maximum allowed speed; this speed depends on both the fMax and the

Initiation Interval (II) specified by the compilation report. This indicator differs from the previous one in that the *throughput unconstrained* is computed using dynamic data obtained from live benchmarks, while the *design throughput* is computed using static data returned by the Intel compiler.

$$X_{design} = BS_{cipher} * (fMax * II) \tag{3.2}$$

Our table reports both the worst design throughput, computed using the worst II value, and the best design throughput, computed using the best II value. The best II value is almost always equal to 1, and was therefore not reported. In general, an OpenCL compiled program will contain multiple kernels, some of which are bounded by containing a loop with a "bad" II, some of which do not contain loops with II greater than 1; therefore, it makes sense to include both in our analysis.

To sum up, the "unconstrained throughput" and "design throughput" indicators both serve to give and idea of which performance we'd be able to extract from our kernels in perfect conditions, albeit with slightly different perspectives.

The data in 3.26 are interesting as they show how, despite the sub-par fMax and high resource consumption values shown in 3.24, the "expensive" blockciphers (Camellia, CLEFIA, SEED) are still virtually capable of reaching very high throughput values.

Table 3.27: Speculative values computed from the results in 3.24 and 3.25, taking available resources into account. For space concerns, “Replication factor” has been abbreviated to “Repl. factor”

Cipher name	Cipher variant	Repl. factor	Throughput unconstrained MB/s	Worst design throughput MB/s	Best design throughput MB/s
AES	single work item, no XTS	1	3,732.00	3,888.55	3,888.55
AES	single work item, only XTS-encrypt	1	-	342.43	2,739.41
AES	single work item, only XTS-decrypt	1	-	1,929.47	1,929.47
Camellia	single work item	1	2,937.69	383.22	3,065.80
CLEFIA	single work item	1	2,653.77	338.65	2,709.20
DES	single work item	1	1,719.59	1,768.88	1,768.88
PRESENT	single work item	1	1,814.77	1,886.06	1,886.06
SEED	single work item	1	3,022.44	384.46	3,075.71
MISTY1	single work item	4	7,662.04	7,875.37	7,875.37
CAST5	single work item	4	8,396.89	8,197.02	8,197.02
HIGHT	single work item	8	10,885.18	11,535.64	11,535.64

Table 3.27 shows the same data as 3.26, this time taking into account the possibility to create multiple alias kernels to use up all the available resources. The table was built using the values from the previous version, multiplied by a *replication factor*. The replication factor was computed by trying to maximize the usage of the scarcest resource among LUT, FF and RAM; furthermore, a slight BRAM-overuse of 120% was allowed.

As we could expect, the fastest blockcipher in this context is HIGHT, since its low area demand allows for a replication factor of 8; indeed, its projected performance improvement is almost ten-fold. CAST5 and MISTY1 take respectively the second and third place, while still being more than twice as fast as the fastest AES implementation. Besides these 3 cases, no other blockcipher attains a replication bonus.

Conclusion

In this paper, we studied the OpenCL implementation of various blockciphers for an FPGA device, and measured how different programming and design techniques affect their performance and resource consumption.

To do so, we first defined a baseline of OpenCL optimization techniques and a set of useful performance metrics; afterwards, we stepped through a series of improvements over our AES implementation, showcasing the effects of different optimization methods.

Finally, we implemented and tested the full range of blockciphers in exam, leveraging the full range of optimiation techniques described in the previous phase. We were also able to identify the limitations of the current runtime environment, and we devised a set of indicators virtually capable of extending our glance beyond said limitations.

3.5 Salient points

We showed how an OpenCL kernel for FPGA benefits from being developed as a single-work-item kernel, and how its performance can be gauged in terms of fMax and Initiation Interval; we also identified useful techniques such as automatic loop unrolling and manual partial loop unrolling, which provide visible performance improvements both alone and combined. A positive performance improvement was also detected when moving the round key from global to local memory, and a possibly greater one when moving again the key from local memory into private memory, made possible by the FPGAs' unique feature to infer registers on-demand. We observed how, in the case of kernels implementing symmetric block ciphers, global memory access coalescing and buffer interleaving are non-issues. We verified how employing a double-buffered scheme allows to partially work around bandwidth limitations in the channel between accelerator device and host, reaching an almost 100% efficient channel usage with no extra cost in terms of FPGA resource consumption. Finally, we posited a series of speculative performance upper-bounds as a starting point for future developments: with the help of a faster host-device channel, we expect an almost ten-fold performance improvement, as allowed by the FPGA area limitations.

3.6 Possible future developments

An interesting direction for future exploration is to acquire a runtime environment with a faster host-FPGA data transfer channel. This would allow to validate the speculations made in Tables 3.26 and 3.27, measure their precision, and possibly unveil some pitfalls which our experiments didn't expose due to the aforementioned bandwidth limitations.

Given the differences and similarities between the programming practices on GPUs and FPGAs highlighted in Table 2.1, we can make the case for a source-to-source compiler, with the purpose of translating data-parallel kernel code, designed to run on GPUs, into task-parallel kernel code, designed to run on FPGAs. As stated in the previous chapters, the FPGA OpenCL compiler does support also traditional data-parallelism; however, both the official documentation, our own studies and third-party research [15] motivate us to prefer a single-work-item implementation.

This newfangled source-to-source compiler should act mainly in 4 directions: First of all, the kernel should be restructured, by enclosing all the original logic into an "outer loop" iterating through all the data blocks normally processed by separate work-items (essentially achieving a serialization of the old work-items and blocks). Next, the special compiler should detect the code fragments responsible for copying global data into local memory for re-use, and replace them with the memcpy-like version used in single-work-item kernels; alternatively, private memory buffers could also be used, provided that the allocated size is not excessive. Afterwards, the source-to-source compiler should generate the data-flow graph for the target kernel, and identify those operation which risk to cause a "same variable load/store" condition; then the compiler could either solve the issue automatically, or just print a warning message. Finally, the special compiler should explore the inner kernel structure, in order to find loop statements to improve by either normal unrolling or using the type of "manual partial unroll" discussed before.

The source-to-source compiler could be designed to parse the data generated by the official Intel report tool, in order to gain further insight of the target kernel's structure.

A small progress has already been made in that specific direction, as some of the graphs and statistics offered in this paper have been obtained by scraping the Intel report data.

Appendix A

Source code

Listing A.1: OpenCL host-side boilerplate code performing vector-addition

```c
const char* programSource =
"__kernel␣void␣vector_sum(__global␣const␣float␣*A,␣␣\n"
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣__global␣const␣float␣*B,␣␣\n"
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣__global␣float␣*C)␣{␣␣␣␣␣␣\n"
"␣␣␣␣size_t␣tid␣=␣get_global_id(0);␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"␣␣␣␣C[tid]␣=␣A[tid]␣+␣B[tid];␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"}␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
;

void vector_add_driver(uint8_t *A, uint8_t *B, uint8_t *C, size_t vector_length) {

    cl_platform_id platform;
    cl_device_id device_id;
    cl_context context;
    cl_command_queue command_queue;

    cl_program program;
    cl_kernel kernel;
    cl_event kernel_complete;

    cl_mem buffer_A, buffer_B, buffer_C;

    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1, &
        device_id, 1);
    context = clCreateContext(NULL, 1, &device_id, NULL, NULL,
        NULL);

    command_queue = clCreateCommandQueue(context, device_id,
        CL_QUEUE_PROFILING_ENABLE, NULL);

    program = clCreateProgramWithSource(context, 1, (const char
        **)&programSource, NULL, NULL);
    clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    kernel = clCreateKernel(program, "vector_sum", NULL);

    buffer_A = clCreateBuffer(context, CL_MEM_READ_ONLY,
        vector_length, NULL, NULL);

    buffer_B = clCreateBuffer(context, CL_MEM_READ_ONLY,
        vector_length, NULL, NULL);

    buffer_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        vector_length, NULL, NULL);

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_A);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer_B);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &buffer_C);

    clEnqueueWriteBuffer(command_queue, buffer_A, CL_TRUE, 0,
        vector_length, A, 0, NULL, NULL);

    clEnqueueWriteBuffer(command_queue, buffer_B, CL_TRUE, 0,
        vector_length, B, 0, NULL, NULL);

    size_t global_work_size = vector_length;
    size_t work_dim = 1;

    clEnqueueNDRangeKernel(
        command_queue,
        kernel,
        work_dim,
        NULL,
        &global_work_size,
        NULL,  // let the implementation choose the local work
           size
        0, NULL,
        &kernel_complete);

    clEnqueueReadBuffer(command_queue, buffer_C, CL_TRUE, 0,
       vector_length, C, 1, &kernel_complete, NULL);

    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(command_queue);
    clReleaseMemObject(buffer_A);
    clReleaseMemObject(buffer_B);
    clReleaseMemObject(buffer_C);
    clReleaseContext(context);
}
```

Appendix B

Text dumps

B.1 board_spec.xml for aclattila_hpc_16_10

```
<?xml version="1.0"?>
<board version="16.1" name="attila_v3_prod">

  <compile project="top" revision="top" qsys_file="none"
     generic_kernel="1">
    <generate cmd="quartus_sh␣-t␣scripts/pre_flow_pr.tcl"/>
    <synthesize cmd="quartus_cdb␣-t␣import_compile.tcl"/>
    <auto_migrate platform_type="a10_ref" >
      <include fixes=""/>
      <exclude fixes=""/>
    </auto_migrate>
  </compile>

  <device device_model="10ax115n4f40i3sg_dm.xml">
    <used_resources>
      <alms num="31307"/> <!-- Total ALMs - ALMs available to
          kernel_system_inst -->
      <ffs num="46988"/>
      <dsps num="0"/>
      <rams num="134"/>
    </used_resources>
  </device>

  <!-- DDR4-2133 -->
  <global_mem name="DDR" max_bandwidth="17067" interleaved_bytes
     ="1024" config_addr="0x018">
    <interface name="board" port="kernel_mem0" type="slave"
        width="512" maxburst="16" address="0x00000000" size="0
        x100000000" latency="240" addpipe="1"/>
  </global_mem>

  <host>
    <kernel_config start="0x00000000" size="0x0100000"/>
  </host>

  <interfaces>
    <interface name="board" port="kernel_cra" type="master"
        width="64" misc="0"/>
    <interface name="board" port="kernel_irq" type="irq" width="
        1"/>
    <interface name="board" port="acl_internal_snoop" type="
        streamsource" enable="SNOOPENABLE" width="32" clock="
        board.kernel_clk"/>
    <kernel_clk_reset clk="board.kernel_clk" clk2x="board.
        kernel_clk2x" reset="board.kernel_reset"/>
  </interfaces>

</board>
```

Bibliography

[1] FIPS PUB 46-3. In *Data Encryption Standard (DES)*. 1999.

[2] IEEE standard for cryptographic protection of data on block-oriented storage devices. *IEEE Std 1619-2007*, pages c1–32, April 2008.

[3] Carlisle Adams. The CAST-128 encryption algorithm. *RFC*, 2144:1–15, 1997.

[4] Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico, and Gerardo Pelosi. Opencl performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives. *Concurrency and Computation: Practice and Experience*, 27(14):3633–3660, 2015.

[5] Altera. *OpenCL on FPGAs for GPU Programmers.*

[6] Altera. *FPGA Architecture White Paper*, 2006.

[7] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, Waterloo, Ontario, Canada, August 14-15, 2000, Proceedings*, pages 39–56, 2000.

[8] BERTEN DSP S.L. *GPU vs FPGA Performance Comparison*, 2016.

[9] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 450–466, 2007.

[10] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns. *CoRR*, abs/1511.02490, 2015.

[11] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E Roback, and James F. Dray Jr. FIPS PUB 197. In *Advanced Encryption Standard (AES)*, page 47. 2001.

[12] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A new block cipher suitable for low-resource device. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, pages 46–59, 2006.

[13] Intel. *Intel FPGA SDK for OpenCL Best Practices Guide.*

[14] Intel. *Intel FPGA SDK for OpenCL Programming Guide.*

[15] Qi Jia and Huiyang Zhou. Tuning stencil codes in opencl for fpgas. In *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*, pages 249–256, 2016.

[16] Khronos OpenCL Working Group. *The OpenCL Specification version 1.1*, 2011.

[17] Hyangjin Lee, Sung Jae Lee, Jaeho Yoon, Dong Hyeon Cheon, and Jaeil Lee. The SEED encryption algorithm. *RFC*, 4269:1–16, 2005.

[18] Fernando Martinez-Vallina and Spenser 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*, page 7:1, 2015.

[19] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[20] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for GPGPU programs. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–11, 2010.

[21] Hidenori Ohta and Mitsuru Matsui. A description of the MISTY1 encryption algorithm. *RFC*, 2994:1–10, 2000.

[22] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, pages 16–31, 2004.

[23] Ilacai Romero Reyes, Irina Fedyushkina, Vladlen Skvortsov, and Dmitry A. Filimonov. Prediction of progesterone receptor inhibition by high-performance neural network algorithm. 7:303–310, 01 2013.

[24] Jie Shen, Jianbin Fang, Henk J. Sips, and Ana Lucia Varbanescu. Performance traps in opencl for cpus. In *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*, pages 38–45, 2013.

[25] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, pages 181–195, 2007.

[26] Gert-Jan van den Braak, Bart Mesman, and Henk Corporaal. Compile-time GPU memory access optimizations. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010*, pages 200–207, 2010.

[27] Kui Wang and Jari Nurmi. Using opencl to rapidly prototype FPGA designs. In *IEEE Nordic Circuits and Systems Conference, NORCAS 2016, Copenhagen, Denmark, November 1-2, 2016*, pages 1–6, 2016.