Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Developing a support for FPGAs in the Controller parallel programming model

Student:
Mr. Gabriel Rodríguez Canal

Advisors:
Dr. Yuri Torres de la Sierra
Dr. Arturo González Escribano

To my family.

Acknowledgements

I want to thank the Trasgo research group for giving me the opportunity to work with them and initiate my research career. I want to especially thank my advisors, Arturo and Yuri, for their implication in my work and all their advise that have made of this work a fruitful, enjoyable experience. Their guidelines have been utterly valuable to grasp the research methodology. I would also like to show my appreciation to my colleagues at the research group, who welcomed me and helped me with their experience.

I would also like to outline the invaluable help of the professor Darío Suárez Gracia, from the University of Zaragoza. Thank you for answering all my questions and sharing your research resources with us.

I want to mention my family as well, who have supported me emotionally during this journey, happy most of the time yet hard and stressful at some moments.

This research has been partially supported by MICINN (Spain), the ERDF program of the European Union and Junta de Castilla y León: PCAS project (TIN2017-88614-R), FEDER Grant VA082P17 (PROPHET project), CAPAP-H6 and MECD (Beca de Colaboración, Spain).

Resumen

La computación heterogénea se presenta como la solución para conseguir supercomputadores cada vez más rápidos capaces de resolver problemas más grandes y complejos en diferentes áreas de conocimiento. Para ello, integra aceleradores con distintas arquitecturas capaces de explotar las características de los problemas desde distintos enfoques obteniendo, de este modo, un mayor rendimiento.

Las FPGAs son hardware reconfigurable, i.e., es posible modificarlas después de su fabricación. Esto permite una gran flexibilidad y una máxima adaptación al problema en cuestión. Además, tienen un consumo energético muy bajo. Todas estas ventajas tienen el gran inconveniente de una más difícil programación mediante los propensos a errores HDLs (Hardware Description Language), tales como Verilog o VHDL, y requisitos de conocimientos avanzados de electrónica digital. En los últimos años los principales fabricantes de FPGAs han enfocado sus esfuerzos en desarrollar herramientas HLS (High Level Synthesis) que permiten programarlas a través de lenguajes de programación de alto nivel estilo C. Esto ha favorecido su adopción por la comunidad HPC y su integración en los nuevos supercomputadores. Sin embargo, el programador aún tiene que ocuparse de aspectos como la gestión de colas de comandos, parámetros de lanzamiento o transferencias de datos.

El modelo Controller es una librería que facilita la gestión de la coordinación, comunicación y los detalles de lanzamiento de los kernels en aceleradores hardware. Explota de forma transparente sus modelos de programación nativos, en concreto OpenCL y CUDA, y, por tanto, consigue un alto rendimiento independientemente del compilador. Permite al programador utilizar los distintos recursos hardware disponibles de forma combinada en entornos heterogéneos.

Este trabajo extiende el modelo Controller mediante el desarrollo de un backend que permite la integración de FPGAs, manteniendo los cambios sobre la interfaz de usuario al mínimo. A través de los resultados experimentales se comprueba que se consigue una disminución del esfuerzo de programación significativa en comparación con la implementación nativa en OpenCL. Del mismo modo, se consigue un elevado solapamiento entre computación y comunicación y un sobrecoste por el uso de la librería despreciable.

Abstract

Heterogeneous computing appears to be the solution to achieve ever faster computers capable of solving bigger and more complex problems in different fields of knowledge. To that end, it integrates accelerators with different architectures capable of exploiting the features of problems from different perspectives thus achieving higher performance.

FPGAs are reconfigurable hardware, i.e., it is possible to modify them after manufacture. This allows great flexibility and maximum adaptability to the given problem. In addition, they have low power consumption. All these advantages have the great objection of more difficult programming with the error-prone HDLs (Hardware Description Language), such as Verilog or VHDL, and the requirement of advanced knowledge of digital electronics. The main FPGA vendors have concentrated on developing HLS (High Level Synthesis) tools that allow to program them with C-like high level programming languages. This favoured their adoption by the HPC community and their integration in new supercomputers. However, the programmer still has to take care of aspects such as management of command queues, launching parameters or data transfers.

The Controller model is a library to easily manage the coordination, communication and kernel launching details on hardware accelerators. It transparently exploits their native or vendor specific programming models, namely OpenCL and CUDA, thus enabling the potential performance obtained by using them in a compiler agnostic way. It is intended to enable the programmer to make use of the different available hardware resources in combination in heterogeneous environments.

This work extends the Controller model through the development of a backend that allows the integration of FPGAs, keeping the changes over the user-facing interface to the minimum. The experimental results validate that a significant decrease in programming effort compared to the native OpenCL implementation is achieved. Similarly, high overlap of computation and communication and a negligible overhead due to the use of the library are attained.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

On this chapter the following aspects are introduced:

- The context and motivation for the work.
- Problem statement.
- The structure of the document.

1.1 Context

The increase in the operating frequency of processors has been the recipe for progress in computing performance during decades, but this is no longer possible due to heat emission. The power dissipated by a processor is proportional to the squared voltage and the frequency. Frequency grows linearly with voltage, so the case for the power is even worse, as it turns out to be proportional to the cubed voltage [49]. This raised the need for other strategies, both software and hardware based, to bypass this limitation.

The traditional sequential approach in which a single CPU core executed one instruction after another evolved to the *concurrent* and *parallel* programming paradigms, which seek the execution of tasks at the same time. This premise can be virtually achieved in concurrent programming with a single core by interleaving small windows of time for the different tasks, creating the illusion of *multitasking*. Parallel computing, on the other hand, achieves true simultaneous execution of tasks by relying on *multicore* processors. These comprise several independent compute units capable of executing one thread each simultaneously. It is common nowadays to find microprocessors with 4 or 8 cores in commodity systems. However, a CPU with N cores will not be able to achieve a speedup of N in a given application, due to data dependencies, serial parts and synchronization costs, partially explained by Amdahl's Law [32]. CPU vendors have worked in the refinement of this approach, creating new architectures and developing new technologies such as hyperthreading [27], capable of increasing the number of virtual cores by scheduling several threads in a given core that are to use a disjoint set of resources. Following this trend, GPUs (Graphical Processing Unit), another microprogrammed architecture initially designed to support the complex calculations of graphical computing, turned out to outperform multicore CPUs thanks to their huge number of simpler compute units, which gives them the name of *manycore*. This ability to better exploit inherent parallelism fostered the development of heterogeneous systems, in which a CPU plays the role of host and the computation is offloaded to a *co-processor* or *accelerator*. This way it is posible to dissect problems and schedule every piece to the most suitable accelerator. Tipically, sequential parts will be solved faster by CPUs, while massively parallel problems fit GPUs better. Despite all the performance benefits of GPUs they come at the cost of high energy consumption. This could be minimised by the use of ASICs (Application Specific Integrated Circuit), that featured just the resources required for a given problem, thus limiting the working die area. This is not a practical solution, since the development of ASICs has cumbersome design costs. Fortunately, reconfigurable hardware gathers both the capability of implementing solutions to different problems of GPUs through programming and the specifity of a circuit adapted to a given problem of ASICs.

FPGAs (Field Programmable Gate Array) are the example of reconfigurable hardware device most used in industry [10, 38]. They have been around since the 80s, when they were used to diminish the cost of prototyping new ASIC designs. They were not capable of holding really complex designs back then, what made them unsuitable for computing. This has changed thanks to the improvements in integration, which has lead to an increase in the density of their logic fabric. They have suffered from heavy and error-prone programming with HDLs (Hardware Description Language) such as Verilog [29] or VHDL [8] in the past, which, on top of that, required deep knowledge of electronics, which hindered their adoption by the HPC [1] community. This has changed in the recent years thanks to the development of HLS (High Level Synthesis) languages and frameworks such as the Intel FPGA SDK for OpenCL by Intel [23] or SDAccel by Xilinx [48], both based on OpenCL, thus improving programmability by replacing HDLs by C-like languages.

1.2 Motivation

The development of HLS tools has eased the integration of FPGAs in heterogenous systems. However, several challenges regarding programmability remain that have to be faced by the programmer. He still has to take care of aspects such as command queue management, kernel launching parameters or data transfers. This results in a cumbersome boilerplate code that severely impacts portability.

The main goal of heterogeneous systems is the exploitation of the particular features of every architecture available to achieve higher performance. This is usually achieved by leveraging the vendor specific tools for every given architecture and having in-depth knowledge of their details. Controllers removes these restriction by providing a uniform user-facing interface to the programmer that enables the use of CPUs and GPUs as accelerators by leveraging the vendor specific frameworks under the hood, thus achieving optimal performance. It also takes care of dependencies, thus easing efficient programmability and allows seamless overlapping of data transfers and computation thanks to its asynchronous policy.

1.3 Problem statement

The aim of this work is to expand the Controllers model by giving support to FPGAs, thus allowing their integration in heterogeneous systems that use the library. Controllers already supported the simultaneous use of CPUs and GPUs. In order to preseve this compatibility, this work should intend for a modular approach.

FPGAs are very different from the traditional microprogrammed architectures. Their ability to adapt to the given problem instead of the other way around and their reconfigurability opens a world of possibilities. However, the particularities of their architecture poses new challenges in the form of different

[1]HPC (High Performance Computing) deals with big scale scientific and engineering problems, such as weather forecasting or simulation of fluid dynamics, seeking a reduction in the time needed to find their solution and giving support to ever increasing input sizes. For that, it involves a combination of research in areas such as compilers, runtime systems, parallel programming and cutting-edge hardware. The last trend for performance are heterogeneous systems, which form the basis of modern supercomputers appearing in the world class list TOP500 [1].

ways of programming at the algorithmic level, a broader set of configurable parameters and compilation times that can last for hours. Despite the huge improvement in programmability achieved by the HLS frameworks, these differences require a further effort to ease their use in a heterogeneous environment. On top of that, memory management is even more critical for these devices due to their slower types of offchip memory. For this reason, a library like Controllers that takes care of data transfers and data dependencies is a must for the proper integration of these devices in heterogeneous systems.

1.3.1 Objectives

The objectives of this work are as follows:

- Study and comprehend the existent works.
- Study the Hitmap [15] and Controllers [30] libraries that lay the foundations of this work.
- Design and implement a prototype of the proposed model.
- Design, implement and execute experimentation cases that enable the validation of the model.
- Obtain results and reach conclusions from the experimentation performed.
- Validate the proposed model.

1.3.2 Budget and planification

This work is the first attempt of the Trasgo research group to include FPGAs in their workflow. As a consequence, some resources have to be acquired before the work starts. In these situations it is necessary to account for the costs of new hardware, software licenses and the time invested in training the involved personnel.

This section presents a rough inital planification and several budget options to get some insight into the costs of this work. Table 1.1 shows the most relevant OpenCL compatible FPGAs in the market at the moment. Many features can be taken into account to choose an option. However, only the most relevant for our purposes have been listed. Features such as the operating temperature or power consumption account for price variability, but they are not that determinant for an initial budget. Finally, the features that made it into the list are:

1. Memory: maximum offchip memory that can be installed on the FPGA board. From fastest to slowest, the three available types are HBM2, DDR4 and DDR3.
2. PCIe: the type of connectivity bus featured by the FPGA board. The highest memory bandwidth will be achieved by x16.
3. LUTs: lookup-tables. This figure expresses the number of configurable logic gates in the FPGA. The higher this number the biggest the design that can be held. Refer to 2.1.1 for more information.

Xilinx Alveo U50 is a cost effective solution that could serve our research purposes, specially compared to the outdated Intel Stratix V. However, since this project will probably lead to further research an Intel Stratix 10 outperforms its competitors because of its 10.2 million of LUTs.

FPGAs need specialised software that performs all the steps related to the process of circuits synthesis. The cost of this software is non negligible and has to be added to the budget. Table 1.2 summaries the

FPGA model	Memory (max)	PCIe	LUTs	Cost
Intel Stratix V [40]	8 GB DDR3	x8	622K	\$6250
Intel Stratix 10 [39]	8 GB DDR4	x16	10.2M	\$10,880
Intel Arria 10 [41]	16 GB DDR4	x8	1.15M	\$5520
Xilinx Alveo U50 [12]	8 GB HBM2	x16	872K	2604.72€
Xilinx Alveo U200 [11]	64 GB DDR4	x16	892K	4382.41€

Table 1.1: Summary of the most relevant features and costs of OpenCL compatible FPGAs.

Software	Cost
Quartus Prime Pro (fixed) [21]	\$3995
Quartus Prime Pro (floating)	\$4995
Xilinx Vivado HL System Edition (fixed) [47]	\$3495
Xilinx Vivado HL System Edition (floating)	\$4295

Table 1.2: Costs of software licenses.

costs of the Intel's and Xilinx's suites, both in fixed and floating license versions. Floating license should be the preferred option, since more than one person might be working on FPGA projects in the future.

Intel offers an alternative to this on-premise solution called Intel Devcloud [3]: 6 months of free access to their HPC cluster featuring Intel Arria 10 FPGAs. More details on the available hardware at 5.5. It offers an adequate experimental environment with homogeneous computing nodes. However, the developer is subject to any change that the Intel team may consider applying to their cluster. For that reason, a local installation is likely in the future.

Table 1.3 shows a tentative coarse task decomposition of the work during the development of this project based on a discussion with the advisors about a possible path to follow and the difficulty of every step. In order for the time estimations to be accurate the profile of every role has to be taken into account, i.e., previous knowledge and abilities. Since both advisors have successfully managed several honours project in the past the times shown in the table are based on previous experience. In the case of the student, it has to be taken into account that this is his first contact with parallel computing, although he has undertaken some projects on its own regarding low level programming and electronics. For this reason, the *learning OpenCL - FPGA* task, which includes traning in the parallel computing paradigm at a conceptual level, is estimated to last 80 hours.

Reading is one of the pillars of research. Given that this is the first contact of the student with research in general and heterogeneous computing and parallel programming in particular, a high proportion of the time allocated for this work will be spent on this task.

Controllers and Hitmap lay the foundations of this work. They are two quite complex tools. However, this work seeks to result in an expansion of Controllers and Hitmap is just an auxiliary tool, from which only a few features are required. This implies that it is requiered to have deep knowledge about the inners of Controllers, but there is no need to reach the same level of comprehension for Hitmap. Knowing the user-facing interface and some implementation details of this library suffice.

The process of implementation can be quite time consuming due to the use of preprocessor macros and the conceptual difficulty associated to metaprogramming. The programmer must be aware of the pieces of code that will be actually expanded in the actual program. Conditional compilation and limited aid from the compiler when debugging turn this into a complicated task, which justify an initial estimation of 100 hours of development.

Experimentation should be a straightforward task once FPGAs are properly integrated in Controllers thanks to the new backend. Since this is the first contact of the student with a project of this kind 30 hours might be a resonable estimate.

Writing this document should be a straightforward task as well, provided that all the results obtained during the work have been properly annotated. Experience shows, however, that students have a hard time writing their first research document, so 40 hours is the prior estimate.

Finally, a certain degree of freedom is advisable in projects management to account for unexpected events. This can be accounted by inflating the estimates of every task. However, since this work is based on research, having some extra time for other side related tasks, e.g. learning some Verilog or CUDA, makes more sense and has been recorded as *Others*.

Role	Task	Time (hours)
Student	Learning OpenCL - FPGA	80
	Reading	100
	Understanding Controllers	60
	Understanding Hitmap	5
	Backend implementation	100
	Experimentation	30
	Writing document	40
	Others	40
Advisor	Meetings	60
	Corrections to the draft	20

Table 1.3: Estimations of the time invested by two roles involved in the work for its development.

All in all, tables 1.1, 1.2 and 1.3 provide the necessary information to estimate the overall cost of the project. For the reasons exposed, the FPGA model chosen for this work is Intel Stratix 10 and a floating license of Quartus is preferred. Regarding the cost of the time invested by the two roles considered a cost of 25€/h for the student and and 50€/h for the advisors is assigned. As a consequence, the total cost of the project, applying an exchange rate of $1.13 = 1€, would be 9628.32€ (FPGA) + 4420.35€ (Quartus) + 11375.00€ (student) + 4000.00€ (advisors) = 29423.67€.

1.4 Document structure

The rest of document is organised as follows: chapter 2 introduces some basic concepts about FPGA architecture and the related work. Chapter 3 describes the final prototype and the previous exploration of FPGA parameters that influenced the decisions taken. Chapter 4 shows the iterative process of implementation of the model. Chapter 5 explains the experimentation conducted over the chosen case studies to validate the proposed model. Finally, chapter 6 discusses the conclusions reached and the possible future research lines.

Chapter 2

Previous concepts and State of the Art

This chapter introduces the following aspects:

- The FPGA architecture and the OpenCL programming model.
- Related work about optimisation of parallel applications.
- The Hitmap and Controllers libraries, developed by Trasgo research group.

2.1 FPGA architecture

An FPGA (Field Programmable Gate Array) is an integrated circuit that can be reconfigured after manufacturing [46, 37]. The most basic models consist of just programmable logic gates and interconnects and I/O ports, which turn them into reconfigurable hardware, as depicted in figure 2.1. It is not uncommon, however, that they also feature other digital components such as digital signal processors (DSPs), dynamic RAM (DRAM) or even microprocessors. These are known as *hard IP* and serve the dual purpose of extending the FPGA capabilities with tested electronics that may otherwise be difficult to implement and saving FPGA surface for other custom non-standard functionality desired by the hardware designer. All of them could be implemented by leveraging the FPGA logic elements, provided that the FPGA is large enough. However, it is often not worth it reinventing the wheel knowing circuits design can be a low productivity and error-prone task. In the same fashion, emulating software libraries, there exists *soft IP*, i.e. HDL (Hardware Description Languages, such as VHDL or Verilog) files usually sold by manufacturers that describe a hardware component that can be instantiated with the FPGA programmable resources. These are not necessarily fixed and allow a certain degree of freedom by exposing configurable parameters, as does the Nios II processor [9].

FPGAs can be manufactured with either fuse/antifuse or SRAM technology. Only the latter is reconfigurable and interesting from the point of view of HPC, so from now on only this kind is considered in this work. This makes FPGAs particularly suitable for applications when data paths are complex and poses a restriction on the utilised surface, since only the targeted elements are needed. This way, it is possible to save energy by leaving the rest of the die idle, as opposed to microprogrammed accelerators, such as GPUs [7].

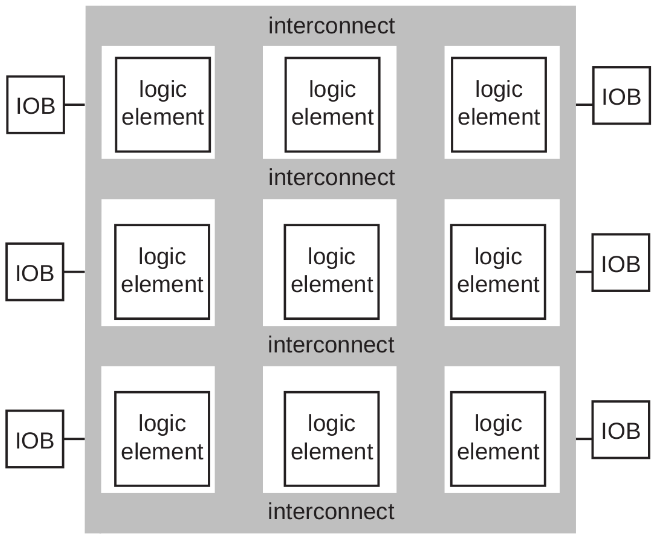


Figure 2.1: Basic architecture of a generic FPGA. Extracted from [46]

2.1.1 Lookup tables

Lookup tables (LUTs from now on) are the configurable elements of the FPGA that provide the combinational logic capabilities. They are formed by SRAM cells (called CRAM by Intel, the C standing for configurable) and a multiplexer. The number of both varies depending on the FPGA model.

Figure 2.2 shows the inners of a LUT from Intel Stratix II (it should be clear that LUTs are a common element in FPGAs regardless of manufacturer, though). CRAM are SRAM cells targeted by the bitstream generated by the compiler, which fixes its value every time the FPGA is reprogrammed. In this example, A, B, C and D are the configurable arguments of the generated function, whilst Y is the output. In the general case, to implement an n-input function an SRAM with 2^n locations is required.

2.1.2 Logic blocks

LUTs are nested withing bigger blocks generally called logic blocks (this denomination varies between manufacturers. Intel, for example, called them Adaptative Logic Modules in Stratix and Stratix II) which always include latches or registers in order to allow sequential circuits to be implemented. This way, FPGAs uses can be expanded further than the mere implementation of combinational circuits, allowing the synthesis of algorithms. Other resources, such as adders, may also be included, as in the case of Stratix II shown in figure 2.2.

The programming process of FPGAs produces a bitstream that is downloaded to the device. This bitstream specifies which elements will be used, their interconnections and the logic functions that are to be implemented. These functions are described by the so called LUT mask i.e. a sequence of bits for every used LUT that describes the state of every CRAM cell. This state will remain fixed until the FPGA is reprogrammed with a different bitstream (or until it is disconnected from power. It is possible for the device to recover that state right after power up thanks to an EPROM that stores the bitstream and reconfigures the FPGA, nonetheless).

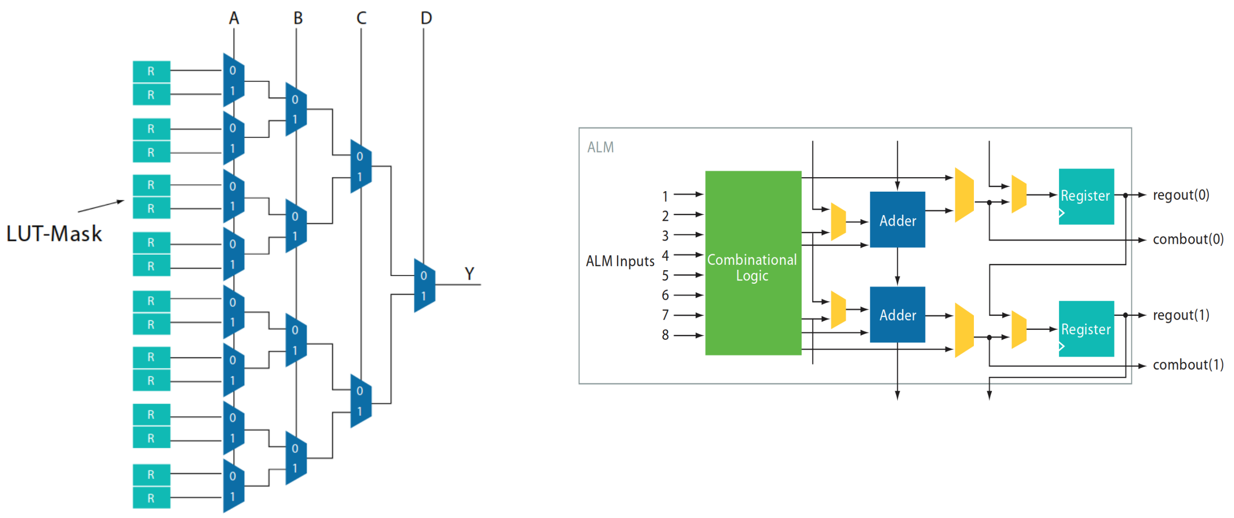


Figure 2.2: Schematic view of a lookup table (left) and a logic block (right). Extracted from [10]

2.1.3 Routing architecture

Routing architecture is another characteristic feature of FPGAs that makes them reconfigurable hardware. The final interconnection grid is the result of an optimisation effort known as *placement and routing* conducted by a CAD tool (Quartus for Intel FPGAs and Vivado for those of Xilinx). *Placement* is the problem of deciding, given the resources the FPGA will need to synthesise a circuit, which part of the circuit it will occupy. This decision is key to achieve high frequency, since FPGAs are not completely regular, taking into account that they are not just composed by the logic fabric, but also feature other resources, such DRAM memory. As a consequence, the distance to this memory, and any other resources that are to be leveraged by the design, must be minimised. *Routing*, on the other hand, deals with the *wiring* of the logic and the periferial I/O blocks by configuring the routing switches depicted in figure 2.3. *Placement and routing* is a highly complex problem that can prolong compilation of kernels to several hours, adding another difficulty to the work with FPGAs.

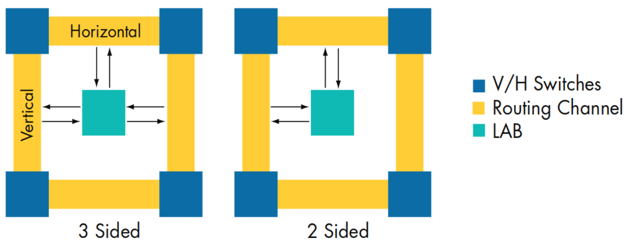


Figure 2.3: Routing architecture of Stratix and Stratix II. Extracted from [10]

2.1.4 SRAM vs. DRAM

In this section the different types of memory available in FPGAs and their particularities will be explained, since a deep understanding is required to achieve high performance leveraging this type of accelerators, due to bandwidth limitations explained later.

In the first place, there is *static random access memory (SRAM)*. This is a low latency RAM that can be used both as logic fabric, as has already been discussed, and memory. The second case will be analysed here, in order to establish a comparison with DRAM. SRAM can be seen as a unidimensional array of memory cells. This makes interfacing with this kind of memory an easy task that can be performed without a memory controller. It can be dual port, i.e. it allows concurrent access to both a reading and a writing agent. On top of that, it is extremely fast, which makes it ideal for caching commonly used data. They are expensive, however, and usually only a few megabytes are available, so it has to be used wisely.

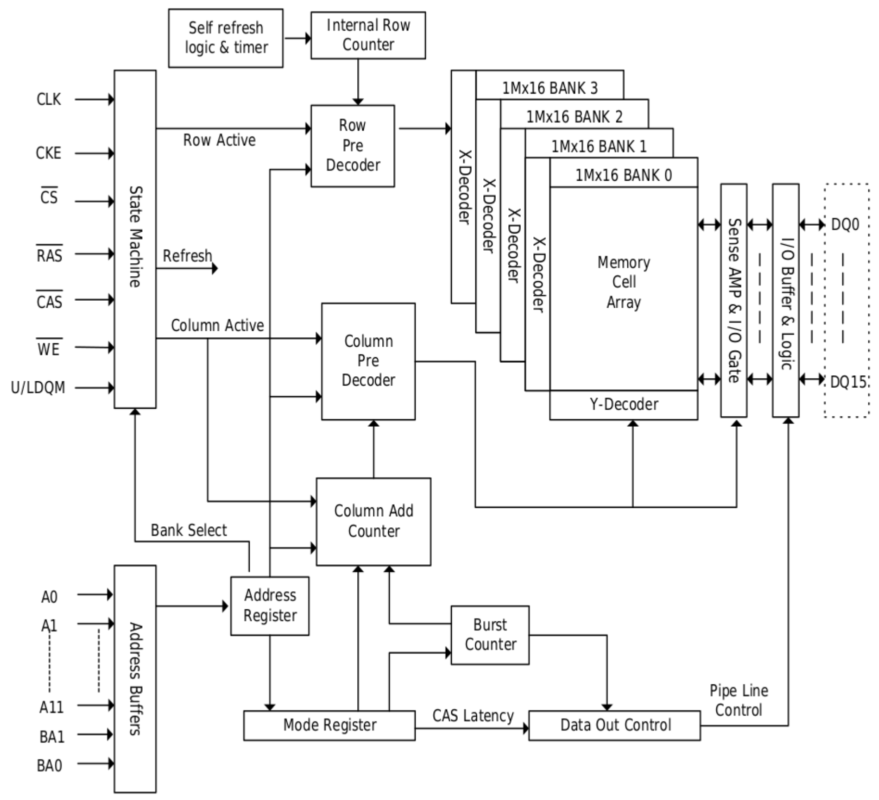


Figure 2.4: Schematic of a DRAM. Extracted from [20]

In the second place, there is *dynamic random access memory (DRAM)* and all its flavours (DDRAM, SDRAM, etc.). For the sake of simplicity, only the generic features of this kind of memory will be explained without going into the specifics of the different specialisations. Unlike SRAM, DRAM is physically assembled as a two dimensional array of memory cells. This might not convey much to the reader unfamiliar with electronics, but it has severe implications on complexity. Not only that, but their cells being based on capacitors adds complexity to its design due to their physical attributes. The value of DRAM cells depend on the charge of the capacitor (when fully charged it acts as a closed circuit, so its value is 0, whilst the opposite happens when it is fully discharged). When a row is to be read, it is copied to a buffer. Capacitors get discharged in the process, losing their information, that will not be recovered until the row is closed. Once that happens the contents of the buffer are copied to the original row, which recovers its values by recharging the capacitors. Moreover, capacitors charge tend to decay with time, so a row that has not been read during the time the capacitor can keep its charge, which is of several milliseconds [34], will lose its value. To prevent this, a refreshing cycle that takes care of this situation by recharging the capacitors at a constant rate established by the manufacturer is implemented. Once the row is available in the buffer, their cells can be accessed there by enabling the CAS (column address strobe) signal [13]. Therefore, rows loaded in this buffer can be accessed up to 98% faster due to the avoidance of row activation latency as can be seen in the last row of table 2.1. As a consecuence, DRAM is more efficiently used when their data is accessed in bursts of contiguous memory locations. This is known as *coalescing* and has a major impact on the way programmers develop their programs. All this complexity justifies the need for a memory controller that hides the details of the component to the hardware designer that is to use it.

Year introduced	Chip size	$ per GiB	Total access time to a new row / column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

Table 2.1: DRAM size increased by multiples of four approximately once every 3 years until 1996, and thereafter considerably slower. Extracted from [34]

2.1.5 Data alignment

Data is stored contiguously in memory and is accessed as explained in section 2.1.4. In order to support values of diverse natures, there exist several basic datatypes with different sizes that, in addition, can be combined to form complex datatypes: structures and unions. Data is retrieved in bursts of fixed width that is established by the width of the bus of the memory controller. This implies that when a given datatype spans a region larger than this width, more than one memory transfer has to be done. Sometimes this inevitable due to a fat complex type, but others this is just due to improper data alignment [6].

The memory controller can only access memory positions aligned to its memory bus width by design. For this reason, data should be aligned to this width to prevent more memory accesses than necessary. The compiler can deal with this problem in many situations, specially with basic data types, but attention should be paid when working with compound data types. Data alignment is mandatory in certain architectures or for a subset of their instructions, as the SSE multimedia instructions of AMD processors. Anyway, it is best practice to keep data aligned to obtain high performance. From now on, we will be considering the case of Intel FPGAs, for which the data bus of the memory controller is 64 bytes wide. This must not be confused with the data bus between the memory controller and the DRAM, since this is not directly accessible to the programmer, as can be seen in figure 2.5.

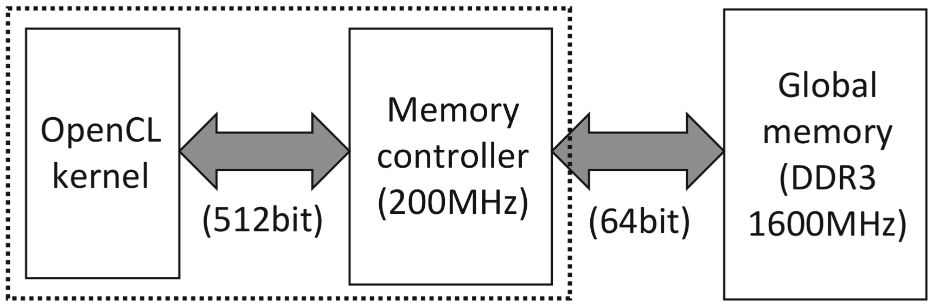


Figure 2.5: Datapath between the global memory and a kernel. Extracted from [44]

An example might clarify what happens when data is not properly aligned. Let s be a structure defined as shown in the code snippet 2.1:

```c
struct s {
    char c;
    int i;
}
```

Listing 2.1: A misaligned structure

s comprises an 1 byte *char* and a 4 bytes *int*. Thus, the total size of *s* should be 5 bytes. However, this might lead to a situation as the one depicted in figure 2.6. The starting address of the structure is 0x36 and the final address is 0x41 (take into account that this is not a fully accurate example, since this address range belongs to the kernel space, and has been designed this way for the sake of simplicity). The first burst that can be transferred lies in 0x00-0x39, but the structure spans further, so a second memory access is needed for such a small datum. This is not the default behaviour of compilers but a technique called *packing* that is useful in embedded applications in which memory resources are scarce. Compilers will usually proceed as shown in figure 2.7, i.e., it will align the structure to the greatest datatype. For that, it will leave as many empty addresses after small datatypes so that the next member gets properly aligned. This empty addresses are called *padding*. In this example, the smallest type of the members in *s* is *char*, followed by an *int* that is unaligned. It is clear three addresses must be left empty after *c* so that *i* becomes properly aligned. This is an inner type of alignment, but the structure itself is not aligned to the 64 bytes boundary, so two accesses are still needed. This can be explicitly requested by the programmer and the compiler will allocate and address multiple of 64 for *s*.

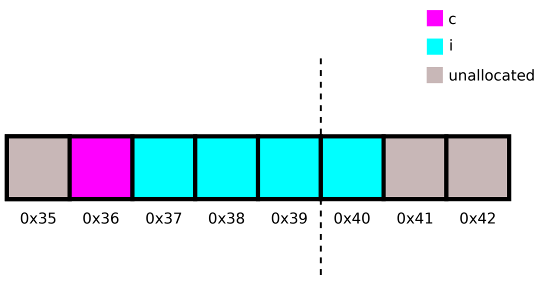


Figure 2.6: Packed misaligned structure. The dashed line indicates which part of the structure lies in the first region transferred and which in the second.

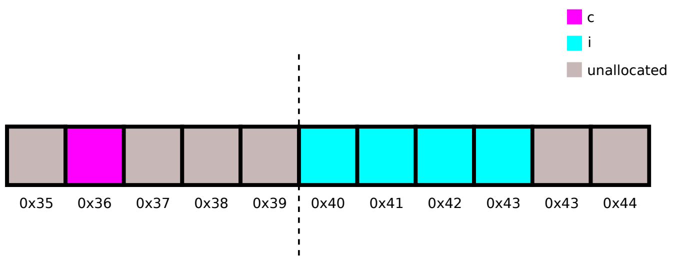


Figure 2.7: Padded misaligned structure. The dashed line indicates which part of the structure lies in the first region transferred and which in the second.

2.1.6 Shift register pattern

Every application should be programmed with efficient memory use in mind, regardless of the architecture of the accelerator, but this becomes specially critical when programming for FPGAs, since they achieve a much lower transfer bandwidth than their counterparts CPUs and GPUs. As an example, the GPU Nvidia Tesla V100 SXM2 can achieve a 897.0 GB/s throughput, whilst the FPGA Intel Arria 10 is limited to 34.128 GB/s. That, added to some potential flaws in the memory controller implemented by Intel implies that the programmer must be aware that he should limit memory access and keep them efficient in order to achieve high performance [52].

Some problems exhibit a property known as *spatial locality*, i.e. adjacent locations are accessed repetitively. This is the case, e.g., of stencil or convolution based problems. For a deeper explanation of one case of each, refer to chapter 5. Take for example, an stencil memory access pattern. As it is shown in figure 2.10 the value of each cell is calculated by applying a linear function over the neighbouring cells in the stencil (green). Notice how every cell composing the stencil will be reused for the calculation of the value of another cell. Since, given a $N \times N$ matrix, this is a $\mathcal{O}(N^2)$ algorithm, $5 \times N^2$ memory accesses have to take place, incurring in overhead. Fortunately, due to the problem structure, a classic electronics construct can help reduce this load: the shift register.

A shift register (figure 2.8) has a fixed number n of stages implemented with flip-flops that get updated at every clock cycle, i.e the datum in flip-flop i passes to flip-flop $i+1$. This behaviour fits the stencil problem if enough memory rows are loaded in a shift register, as illustrated in figure 2.10. In the case of a 2D 5-point stencil at least 2 rows and a cell must be loaded in the shift register to get any benefit from exploiting this pattern, as shown in figure 2.9 in red. It is desirable to have as many preloaded memory positions in the shift register as possible (in pink), as these equal the number of steps that can be calculated without retrieving new values from memory. This way accesses are reduced to N^2. However, although the improvement is in terms of a multiplicative constant, accesses are coalescable, since memory positions are adjacent, and this leads to great speed improvements. In the case of a 1-D stencil, this is much more noticeable, since the complexity of the algorithm is modified: given an M length stencil and an N length array, accesses improve from $N \times M$ to just N [25].

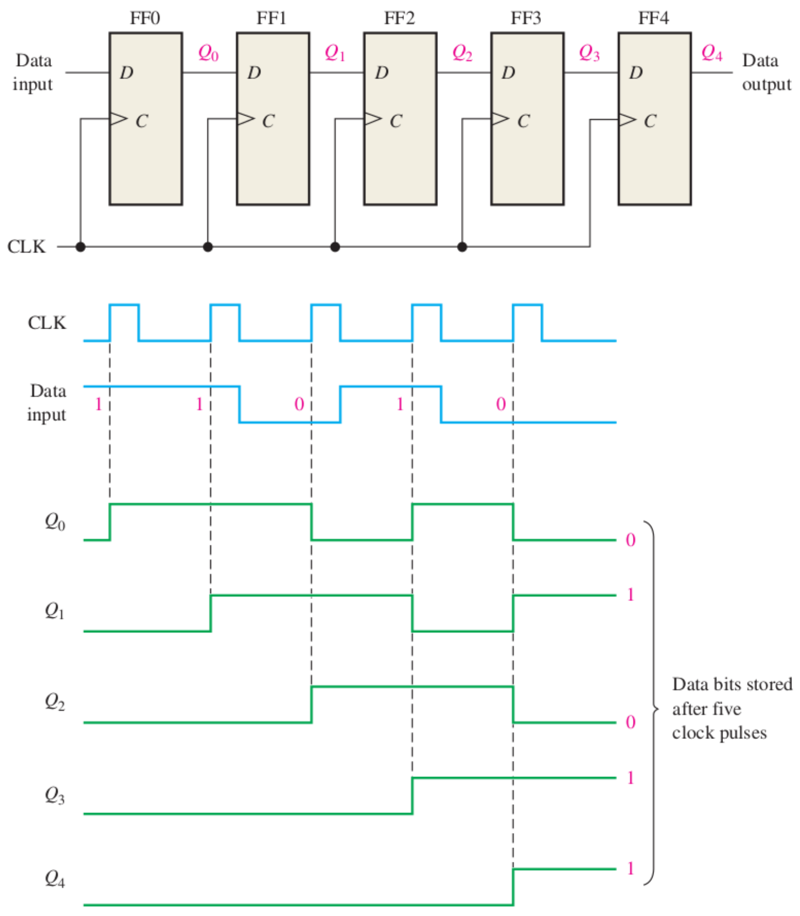


Figure 2.8: Schematic and timing diagram of a serial shift register made out of D flip-flops. Extracted from [14].

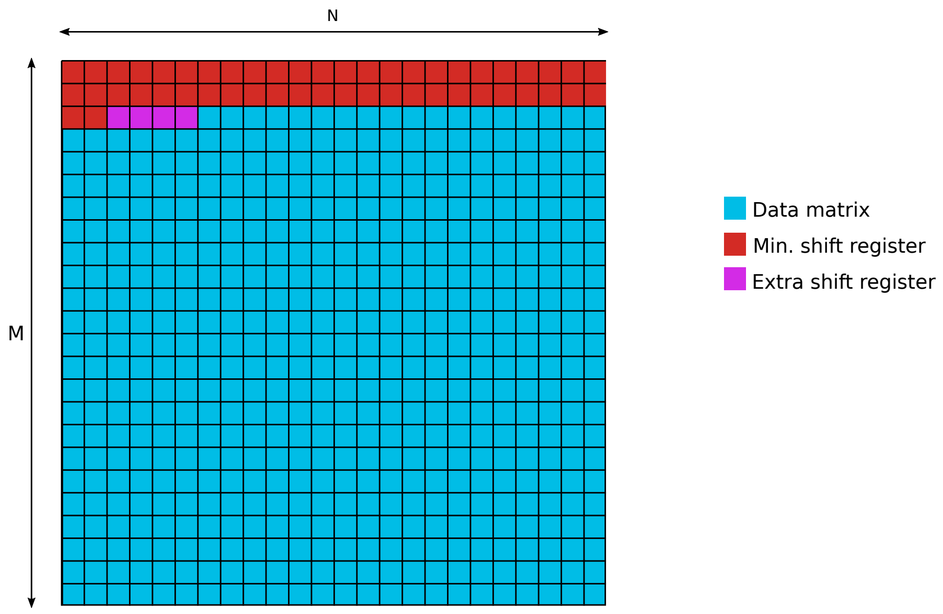


Figure 2.9: Shift register pattern applied to bidimensional data.

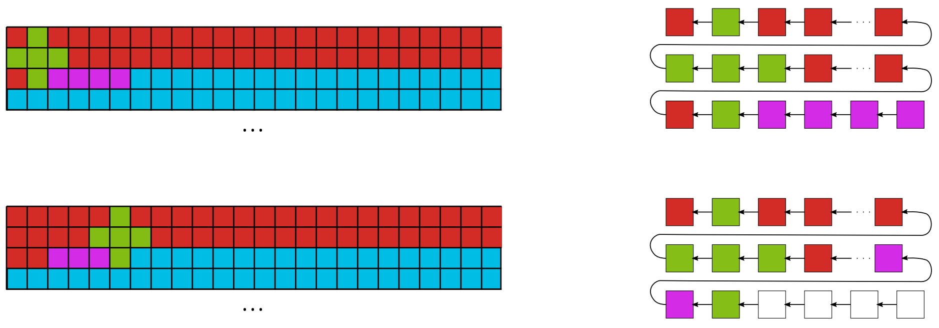


Figure 2.10: First and final step of an iteration of a stencil calculation.

2.2 Related work

This section gives an overview of the related work. First, some insight is be given into the previous work developed by the Trasgo research group that has been leveraged in this work. After that, the basics of the OpenCL model are explained to enable the comprehension of the rest of this document. Finally, Controllers is compared with other heterogenous programming models.

2.2.1 Hitmap

The Hitmap library [15] offers automatic techniques of partitioning and mapping of data, in an efficient and configurable way, in run time. This library defines an abstraction interface and a plug-in system that encapsulates regular and irregular techniques, helping to generate code independent from the chosen mapping function. Hitmap supports partitioning (*tiling*) of sparse or compact array distributions. This partitioning enables the implementation of data and tasks parallelism according to the SPMD (Single Program Multiple Data) model. This library offers functionalities to create, manipulate, distribute and communicate these partitions (*tiles*) and hierarchy of partitions.

The Hitmap library introduces the following concepts:

Signature: A signature S is defined as a tuple of three integer elements representing a subspace of array indices in a one-dimensional domain. It resembles the classical Fortran or MATLAB notation for array-index selections. The cardinality of the signature is the number of different indices in the domain

$$S \in \text{Signature} = (\text{begin} : \text{end} : \text{stride})$$
$$\text{Card}(s \in \text{Signature}) = \lfloor (\text{s.end - s.begin})/\text{s.stride} \rfloor .$$

Shapes. We define a *Shape h* as a n-tuple of signatures. It represents a selection of subspace of array indices in a multidimensional domain (multidimensional parallelotope). The cardinality of the shape is the number of different index combinations in the domain

$$h \in \text{Shape} = (S_0, S_1, S_2, \ldots, S_{n-1})$$
$$Card(h \in \text{Shape}) = \prod_{i=0}^{n-1} \text{Card}(S_i).$$

Tiles. We define a *Tile* as an n-dimensional array. Its domain is defined by a shape, and it has a number of elements of a given type, depending on the programming language chosen

$$\text{Tile}_{h \in \text{Shape}} : S_0 \times S_1 \times S_2 \times \ldots \times S_{n-1} \longrightarrow <\text{type}>.$$

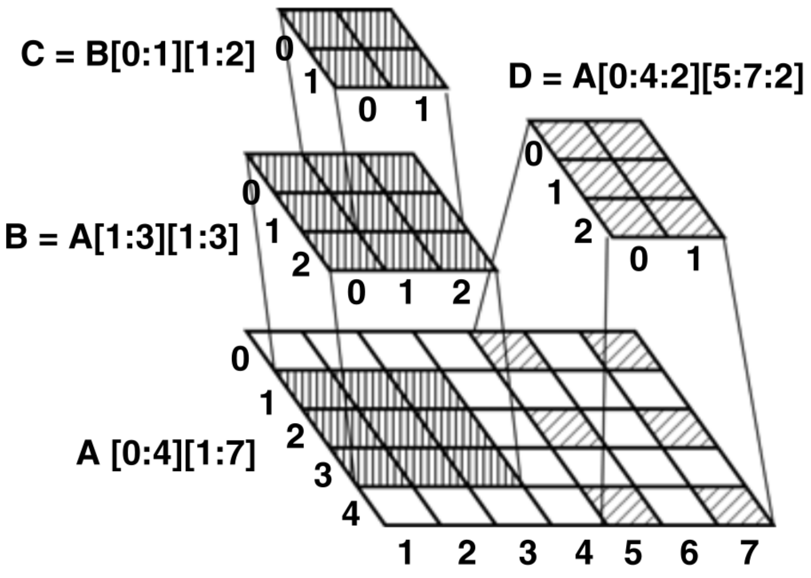


Figure 2.11: Tiling creation from an original array. Extracted from [15].

Hitmap supports three sets of functionalities:

- *Tiling functions.* Definition and manipulation of arrays and tiles, in a tile-by-tile basis. These functions can be used independently of the others, to improve locality in sequential code as well as to generate data distributions manually for parallel execution.
- *Mapping functions.* Data distribution and layout functions to automatically partition array domains into tiles, depending on the virtual topology selected. These functions are oriented to data and task distribution on parallel environments. The input needed at this point are the virtual topology and layout functions to be used, and the data structure to be distributed. These functions return i) the ranges of the tiles that need to be created (using the tiling functions), ii) the mapping between tiles and virtual processors, and iii) the neighbour information, encapsulated in a single structure.

- *Communication functions.* Creation of reusable communication patterns for distributed hierarchical tiles. These functions are an abstraction of a message-passing model to communicate tiles among virtual processors, and may be used with the mapping information (mapped tiles, neighbourhood information, and virtual topology), to create mapping-dependent communication patterns. They return a handler that can be used to repeatedly communicate tiles among processors.

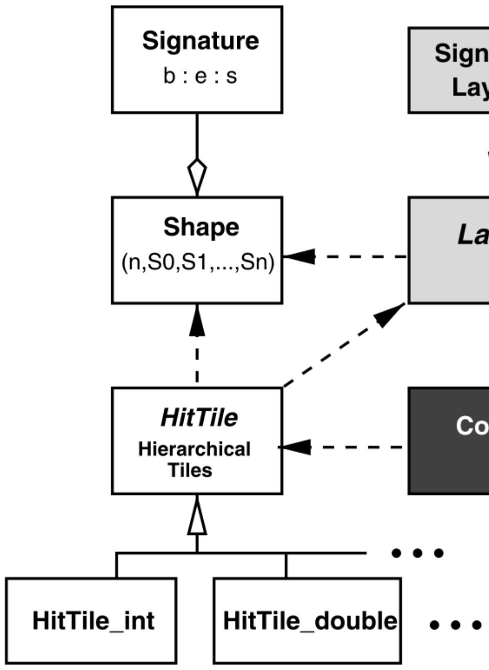


Figure 2.12: UML diagram of the Hitmap library architecture. Extracted from [15].

2.2.2 Controllers

The *controller* model [30] introduces a simplified way to program application that can exploit heterogeneous computational platforms including accelerators or/and multi-core CPUs. Its architecture is represented in figure 2.13. The device controllers coordinate the execution of series of kernels. These kernels are declared as functions, that are managed by the controller entities. Controllers automatically manages the two main concepts used in a program that exploits accelerators.

1. **Kernel management**, including the kernel launching and configuration. The controller manages the deployment/execution of sequences of kernel functions in the computational device associated to the controller. The controllers can include policies to exploit concurrent kernel execution techniques, interleave computations with communications or reorder the sequence of kernels. The kernel configuration is the selection of specific configuration parameters for the kernel launching that can be associated to a particular kernel and computational platform.

2. **Data management**, including the data transfers carried out across the memory hierarchies of the host and the accelerators and the abstraction used to access data elements independently of the target device, the threads indices space or the data layout.

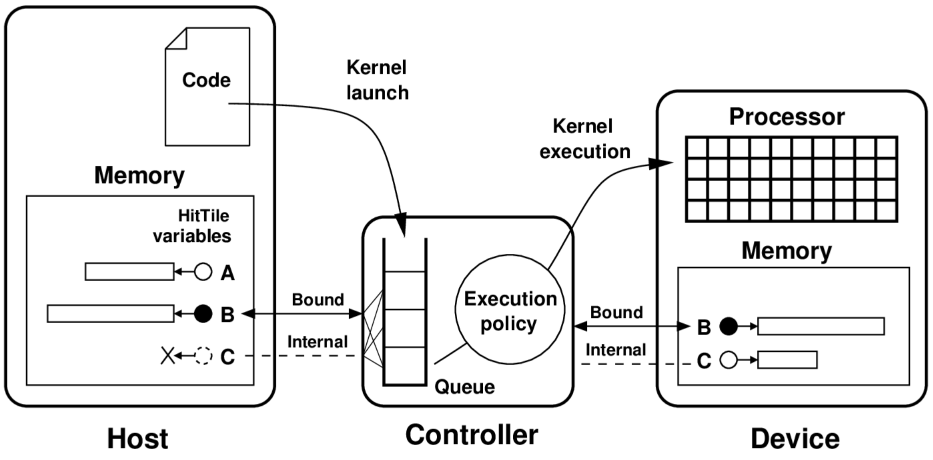


Figure 2.13: Diagram of the controller model architecture. The kernel-launching requests can be enqueued. The controller entity manages the execution of enqueued kernels, and for bound variables, the data transfers between memory spaces. In the figure, the host variable A is not bound to the controller. Variable B is a bound variable, with a duplicated image for the data in the device memory. Variable C is an internal variable of the controller, defined in the host, but allocated only in the device. Extracted from [30].

Rules for asynchronous execution

The rules applied at run-time to decide which operations should wait to be ready for execution, and which ones can be started, executed asynchronously, and potentially be overlapped with previous or subsequent operations are based on data dependencies.

The decisions of which kernels should be or not be executed concurrently to improve the overall performance of the application is a problem orthogonal to the overlapping of data transfers with kernel or host-task execution, and it is beyond the scope of this work. In this proposal, the order of the execution of kernels is strictly preserved. Similarly, requests for the execution of host-tasks are also executed in order, although they can be overlapped with kernels if dependencies allow it. Data transfers can be overlapped with the execution of both kernels and host-tasks.

The internal rules that decide when a request can be safely started have been designed by studying the dependencies between the different types of requests and between the input/output role of their parameters. Allocate/deallocate and wait operations have simple rules due to the synchronisations required by their semantics. Each data structure can have two memory images, one in the host and one in the device. The kernel, host-task and data transfer requests have been analysed considering them as a multiple-reader/multiple-writer problem. The dependencies generated by this scheme are depicted in 2.14.

Queue management policy

Controllers implements two policy modules: synchronous and asynchronous execution. The asynchronous policy module pops tasks from the queue, but it delegates to the backend the application of the rules depicted in 2.14. Thus, different mechanisms can be exploited on different backends, using the particular primitives and resources provided by the specific programming model. The backend methods use the information provided in the request parameters, and in the associated tiles, to evaluate the rules and transform the request into calls to the particular programming model for the specific device. The synchronous policy works with the same model, but always introducing a dependency with the previously

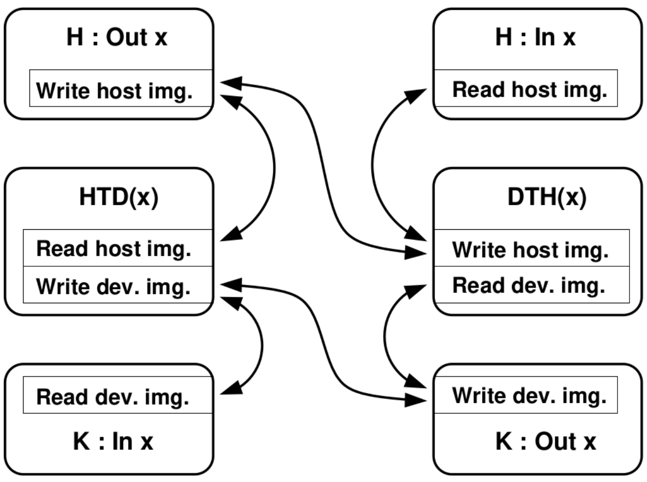


Figure 2.14: Dependencies between request types. Rounded boxes identify request types, using the x data structure as parameter. We distinguish between K or H requests that use x with an input or output role. For clarity, the Wait and Alloc/Free operations are skipped in the figure. Inside each box we represent in small boxes whether the request reads or writes in the host or device memory image of x. Arrows express dependencies between request types implied by the use (read or write) of the memory images of x. Requests that are not linked with arrows can be executed concurrently. Remind that this model does not consider the case of several concurrent host-tasks.

evaluated request to sequentialise them.

In the Controllers model a kernel is declared using two primitives. The first is *CTRL_KERNEL_PROTO*, that declares a prototype for all the implementations of a given kernel. See lines 1-5 in figure 2.2. It declares the number of implementations available and for which backend or devices are those implementations designed. This is used by the Controller to locate, at run-time, the best available implementation of a kernel for the chosen device. In the example, there is only one implementation declared. The GENERIC keyword indicates that it is usable on any backend. Other keyword names are associated to the specific types of backend or device. The rest of the prototype is the description of the kernel parameters, including their input/output roles, types and names. INVAL indicates a value parameter. IN, OUT, or IO (in/out) indicate the role of a HitTile received as reference.

Each kernel implementation is declared by using the primitive *CTRL_KERNEL*. See lines 7-17 in figure 2.2. The first parameters are the kernel name, and the type of implementation. After the declaration of the kernel parameters, the code is included in a structured block. This particular example computes a *saxpy* operation, fused with the application of the square root on each element of the result. The implementation is a generic fine-grain data-parallel specification, that can be automatically coarsened and adapted by the Controller to the proper task granularity of different types of devices, such as GPU, or sets of CPU cores.

```
 1 CTRL_KERNEL_PROTO ( saxpy_sqrt , 1 , GENERIC ,
 2         3,
 3         IVAL , int , alpha ,
 4         IN , HitTile_float , matrix_x ,
 5         IO , HitTile_float , matrix_y ) ;
 6
 7 CTRL_KERNEL ( saxpy_sqrt , GENERIC ,
 8         int alpha ,
 9         KHitTile_float matrix_x ,
10         KHitTile_float matrix_y ,
11 {
12     const int row = threadId . x ;
13     const int col = threadId . y ;
```

```
    hit ( matrix_y , row , col ) = sqrt (
            alpha * hit ( matrix_x , row , col ) +
            hit ( matrix_y , row , col ) ) ;
} );
```

Listing 2.2: Example of a kernel prototype and its implementation, using the Controller library

```
int main(int argc, char *argv[]) {

    Ctrl_Thread threads;
    Ctrl_ThreadInit(threads, SIZE, SIZE);

    __ctrl_block__(1)
    {
        PCtrl ctrl = Ctrl_Create(CTRL_TYPE_FPGA, CTRL_POLICY_ASYNC, DEVICE, &PLATFORM);

        HitTile_float matrix_a = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE, SIZE) );
        HitTile_float matrix_b = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE, SIZE) );
        HitTile_float matrix_c = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE, SIZE) );

        HitTile_float matrix_tmp = hitNewTile(float, hitNewShapeSize(SIZE, SIZE));

        Ctrl_HostTask(ctrl, Init_Tiles, matrix_a, matrix_b, matrix_c);

        Ctrl_Launch(ctrl, Mult, threads, matrix_c, matrix_a, matrix_b, SIZE, SIZE);


        Ctrl_HostTask(ctrl, Init_Tiles, matrix_a, matrix_b, matrix_c);

        // Matrix Mult: Main loop
        // Clock: Synchronise and start measuring
        Ctrl_MoveTo(ctrl, matrix_a, matrix_b, matrix_c);

        for (int i = 0; i < N_ITER; i++) {
            if ((i % 2) == 0) {
                Ctrl_Launch(ctrl, Mult, threads, matrix_c, matrix_a, matrix_b, SIZE, SIZE);
                Ctrl_MoveFrom(ctrl, matrix_c);
                Ctrl_HostTask(ctrl, Host_Compute, i, p_sum, p_res, matrix_c, matrix_tmp);
            } else {
                Ctrl_Launch(ctrl, Mult, threads, matrix_b, matrix_a, matrix_c, SIZE, SIZE);
                Ctrl_MoveFrom(ctrl, matrix_b);
                Ctrl_HostTask(ctrl, Host_Compute, i, p_sum, p_res, matrix_b, matrix_tmp);
            }
        }

        Ctrl_GlobalSync(ctrl);
        // Clock: synchronise and stop measuring

        Ctrl_Free(ctrl, matrix_a, matrix_b, matrix_c);
        hit_tileFree(matrix_tmp);

        Ctrl_Destroy(ctrl);
    }

    ...
}
```

Listing 2.3: Snippet of the main code of a Matrix Multiplication programmed using the Controller library (asynchronous)

2.2.3 OpenCL

OpenCL (Open Computing Language) [16] is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. As such, it does not provide an implementation for the different architectures, but they are developed by the vendors who decide to give OpenCL support to their devices, as in the case of Intel with their Intel FPGA SDK for OpenCL.

Platform model

The platform model of OpenCL consists of a host that is connected to one or several accelerators (devices in OpenCL terminology). Devices are formed by compute units and compute units, at the same time, by processing elements. Once again, it is up to the vendors how to map this concepts to their architecture. In the case of FPGAs compute units and processing elements tend to be considered the same, since a compute unit is always mapped to a pipeline.

The role of the host in this model is that of submitting commands to the devices that will be used to perform computations.

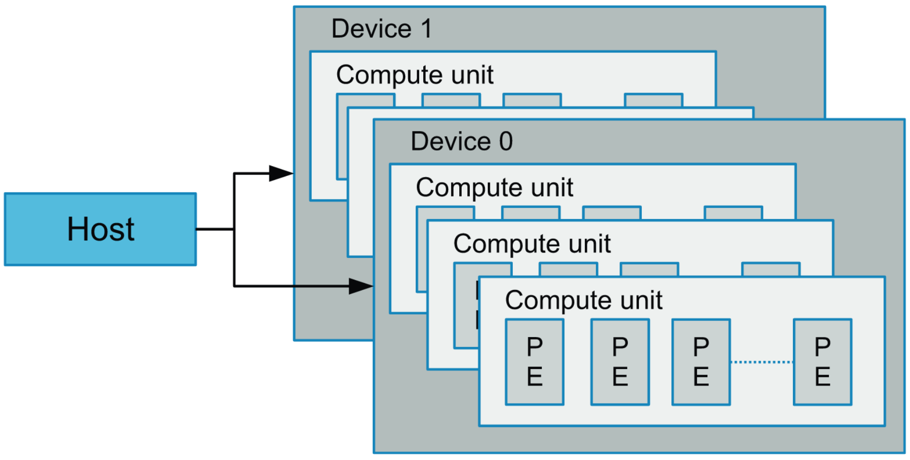


Figure 2.15: Depiction of the OpenCL platform model. Extracted from [26].

Execution model

OpenCL applications comprise two parts: *host program* and *kernels*. The host program is executed by the host and contains the entry point of the application. It takes care of orchestrating all the devices and performing tasks such as managing data transfers between host and devices, submitting commands to the accelerators or processing the events they generate. On the other hand, the *kernel* is the function that is executed on one or more accelerators and conveys the computational load of the program.

The core feature of kernel execution in OpenCL is the *index space*. A given instance of a kernel, named *work-item*, is associated to a point in that index space with its own global ID, equivalent to the so named *thread* in other models such as CUDA. Although every work-item executes the same code, frequently they operate on different data and can even follow different execution pathways, leading to a concept known in parallel programming as *branch divergence*.

Work-items are organised into *work-groups*, which are a more coarse-grained decomposition of the index space and have their own ID. Work-items are assigned a unique local ID within the work-group as well. This makes possible to identify work-items either by their global ID or by a combination of their work-group ID and their local ID.

According to the OpenCL specification every work-item runs on a single processing element, whilst every work-group is mapped to only a compute unit. Since FPGAs are not microprogrammed, PEs and CUs are equivalent concepts when working with these devices, i.e, work-groups run, indeed, on a single pipeline (compute unit), but their work-items will execute through all stages of the pipeline, instead of sequentially executing all the instructions of the kernel code as they would on a CUDA core of a Nvidia GPU (processing element).

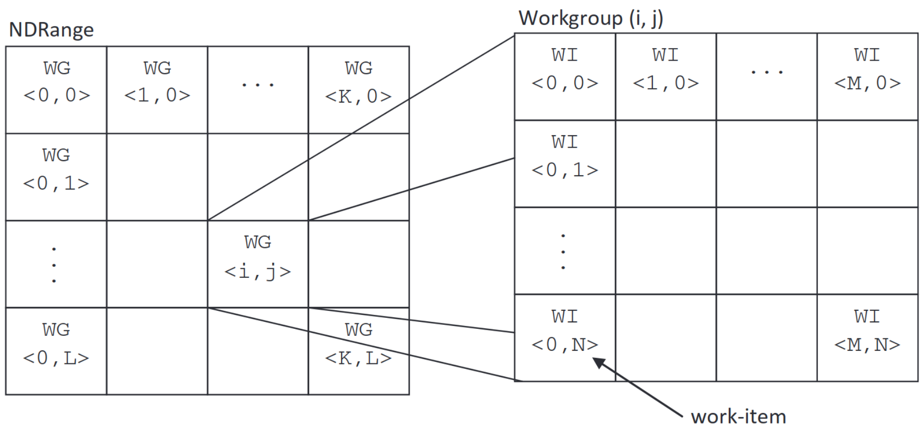


Figure 2.16: Depiction of the OpenCL execution model. Extracted from [26]

The execution of kernels takes place on a given context, which gives access to the host to the following resources:

- **Devices**: the collection of accelerators used by the host.
- **Kernels**: the OpenCL functions that run on OpenCL devices.
- **Program objects**: the program source and executable that implements the kernels.
- **Memory objects**: a set of memory objects visible to the host and the OpenCL devices.

Memory model

OpenCL defines four memory address spaces that can be accessed by the work-items:

- **Global memory**: accessible for every work-item in every work-group. It is usually the slowest type of memory and tends to be off-chip (off-chip DRAM in FPGAs).
- **Constant memory**: this is a special region of global memory that remains constant during the execution of the kernel, acting as a sort of caché.
- **Local memory**: shared among work-items in the same work-groups. It is usually implemented in faster memory than global memory (on-chip SRAM in FPGAs).
- **Private memory**: accessible only by a given work-item. It is frequently the fastest type of memory and is usually mapped to registers.

Programming model

OpenCL supports the data parallel programming model and the task parallel programming model. The first one distributes computation among a set of work-items organised as has been explained in the execution model. This type of kernels are called ND-Range kernels. The second model relies on a single work-item to perform the computations, which gives place to the so called Task kernels.

The programming model of OpenCL defines two domains of synchronisation: work-items in a single work-group and commands enqueued to command queues in a single context. The first is performed using a work-group barrier, whilst the second relies on command-queue barriers and events, the approach leveraged in Controllers.

2.2.4 Other programming models

There are several projects that intend to develop heterogeneous programming models seeking the same goals than Controllers, such as EngineCL [31], the University of Thessally's [33], OmpSs@FPGA [5] or FPM [45].

EngineCL and the University of Thessally's model improve programmability in heterogeneous systems by leveraging OpenCL under the hood for every supported architecture. This can lead to higher overheads than using the vendor specific tools such as CUDA for Nvidia GPUs. On top of that, EngineCL depends on C++ fancy features such as variadic templates, initialiser lists or rvalue references, thus denying to the programmer the possibility of integrating the tool with applications coded in the C language. Besides, the geometry for kernel execution must be manually specified by the programmer with the University of Thessally's model through the *workers()* and *groups()* classes.

OmpSs@FPGA and FPM do have vendor specific backends for every supported architecture, thus avoiding the performance issues suffered by the previous models. They rely on non-standard directives to express parallelism and dependencies and a source-to-source compiler that translates them into runtime system API calls, a common feature with the University of Thessally's model. Forcing the programmer to include a source-to-source compiler in his software stack might cause trouble when the model is to be included in an environment that implements another compiler specific syntax.

Controllers decreases programming effort while keeping the overhead resultant from the use of the library to the minimum. To that end, it features vendor specific tools. It is designed to be compatible with the C99 standard by avoiding the use of the aforementioned C++ features, which are replaced by preprocessor macros, thus allowing the generation of portable code in both languages and keeping the model compiler agnostic. Finally, the geometry for kernel execution can be automatically adjusted by Controllers in run time thanks to the characterisation of the kernel.

2.3 Conclusions

The first part of this chapter introduced the FPGA architecture. It gave some insight into the complexity of this type of accelerators and explained the inner mechanisms that allow reconfigurability. Some details about memories featured by FPGAs and how to overcome their limitations were also given.

In the second part the related work was presented. A brief description of the previous work of Trasgo research group relevant to this work, namely Controllers and Hitmap, was given to ease the comprehension of the rest of the document. Finally, an analysis of several state of the art heterogeneous models was made in order to establish a comparison with Controllers.

Chapter 3

Description of the solution

This chapter introduces the following aspects:

- The configurable parameters in FPGAs that are to be supported by the Controllers model.
- An extension of the Controllers model that allows the execution of kernels in FPGAs with automatic overlapping of communication and computation operations.

3.1 Approach to the solution

So far HPC has been based on the exploitation of microprogrammed architectures such as CPUs and GPUs, which have something in common: they are fixed by the manufacturer. Reconfigurable computing has broken this paradigm by allowing the programmer to design his own architecture. Although this has the benefit of adapting the hardware to the problem, resulting in an increase in performance in certain applications and significant savings in power [4] it comes at the cost of more difficult programming. Although this has been alleviated by HLS languages, mainly OpenCL, in the recent times, leading to an increase in productivity, there are still plenty of tunable parameters. Since the success of a given FPGA implementation relies on these parameters, a previous exploration must be conducted in order to find how we can give support for them in our model. Take into account that all the work shown in this document has been undertaken using an Intel FPGA, so this section will reflect the possibilities offered by the software stack of this manufacturer. Xilinx FPGAs are programmed in OpenCL as well, so no major changes would have to done to integrate them apart from the syntax to configure these parameters.

This section shows the proposal of a new backend for the Controller library that gives support to FPGAs. This solution reduces the programming effort, allows automatic management of dependencies and seamless overlapping of data transfers and computations thanks to its asynchronous policy. It keeps the changes to the user-facing interface to the minimum while adding support for the studied parameters that are most relevant for performance.

3.2 Configurable parameters

There are two types of configurable parameters: compiler parameters and kernel parameters. Compiler parameters must be fixed by the programmer prior to the kernel compilation. Otherwise, the default value will be used. On the other hand, kernel parameters may not be configured by the programmer and the compiler will adapt its value depending on the data access patterns.

3.2.1 Compiler parameters

The Altera Offline Compiler (AOC from now on) presents several parameters to the programmer that can potentially increase performance.

- **no-interleaving**: by default, OpenCL allocates buffers interleaved in memory, as shown in figure 3.1. This is generally the configuration that guarantees more balanced memory accesses. However, there might be situations in which it is known beforehand that the buffers allocated will be accessed independently and there are enough memory banks to map every buffer to a different bank. This strategy might increase the memory bandwidth, which will result in a performance increase. This situation is shown in figure 3.1.

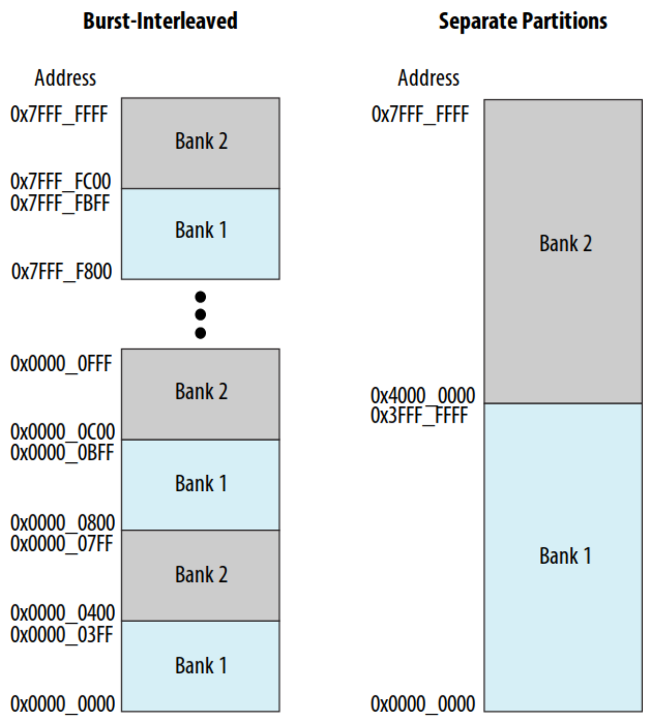


Figure 3.1: Memory interleaving (left) vs no interleaving (right). Extracted from [24].

- **const-cache-bytes**: establishes the size of constant memory (implemented with *on-chip* memory elements, eSRAM and M20K). It works as a ROM memory.
- **fp-relaxed**: allows the compiler to relax the order of arithmetic operations, possibly affecting the precision. This option might increase the FPGA working frequency by simplifying the generated hardware. As precision might be compromised, the current problem must be tolerant to some arithmetic error.

3.2.2 Kernel parameters

This section introduces the most relevant kernel parameters specified in the kernel source code. They are provided as attributes by using the annotation __attribute__() and allow the configuration of local memory and the kernel pipeline.

Local memory

Local memory is by far the most configurable element when programming the FPGA using OpenCL. It can lead to outstanding performance improvements.

- **numbanks(N)**: fixes the number of banks for a given local memory.
- **bankwidth(N)**: changes the width of the bank in bytes (N).
- **singlepump / double pump**: this is a technique widely used in electronics for virtually doubling available ports by doubling the operating frequency of the resource. By default, each local memory replicate has two physical ports. The double pumping feature allows each local memory replicate to support up to four physical ports. This is achieved by using the available ports both in the high and low edge of the global clock. This is possible because the frequency of the memory block doubles that of the global clock.

  Figure 3.2 shows how double pumping is implemented. There it is clearly shown how the two multiplexers serve the purpose of enabling the ports from high or low edge clock, as the selection bit is connected to the global clock and the M20K block operates at 2× frequency of the global clock.

  This is known in electronics as a resource sharing technique. As such, it reduces usage of local memory at the expense of a higher logic utilisation. It might reduce kernel clock frequency as well, so it should be used carefully.

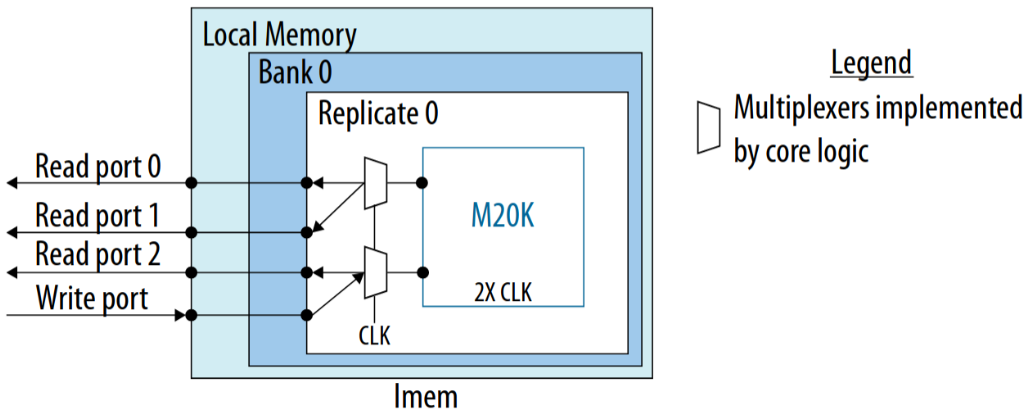


Figure 3.2: Schema of double pumping. Extracted from [24].

- **numreadports(N)**: number of read ports of the memory implementing the variable.
- **numwriteports(N)**: number of write ports of the memory implementing the variable.
- **merge('label', 'direction')**: joins memories with the same label according to the direction specified by 'direction'. Suppose two arrays **a** and **b** with the same label. If the 'depth' direction is passed to merge then **a** and **b** are mapped as shown in figure 3.3 and accessed separately. If, on the other hand, the direction is 'width', i-th elements of both arrays are accessed at the same time.
- **bank_bits**($b_0, b_1 \ldots, b_n$): memory banks are generated according to the lowest array index. This can be modified with this attribute, choosing the addressing bits.

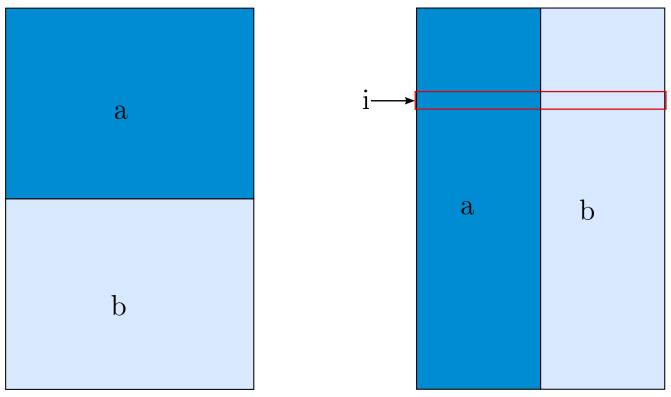


Figure 3.3: Depth-wise merge (left) and width-wise merge (right).

3.2.3 Pipeline configuration

OpenCL kernels are implemented as pipelines in FPGAs. This implies that code is executed in a sequential way, which seems to collide with the parallel programming paradigm leveraged to increase performance with other architectures, such as GPUs. As FPGAs are reconfigurable hardware this is easily adapted to explode parallelism. There are two ways to do this: pipeline replication and pipeline vectorisation.

Pipeline replication

A single kernel pipeline maps to the concept of compute unit in OpenCL. A compute unit is capable of executing one workgroup at a given time. Once all the stages of the pipeline have been filled every cycle a work-item completes. A new workgroup cannot get into the pipeline until the first stage of the pipeline becomes free, which could lead to suboptimal performance in ND-Range kernels (this is not the recommended type of kernels for FPGA, however, since single-work-item kernels work better in general according to Intel). This can be bypassed by replicating the compute unit, so more than one work-group can be executing at the same time. There is no restriction on how many compute units can be instanced, but the available area in the FPGA. Massive replication of compute units is not recommended, since memory bandwidth could decrease because different workgroups will be competing for the same global memory resources. Figure 3.4 shows the performance of the unidimensional Jacobi, an unoptimised stencil kernel similar to Hotspot (5.2) whose all memory accesses are to global memory. It is clear that increasing the number of replicates causes memory contention, which eventually leads to poor performance.

Pipeline vectorisation

Vectorisation allows several workitems in the same workgroup to execute in parallel in the same pipeline. Workitems will execute simultaneously in the same stage of the FPGA during their advance through the pipeline. There is an imposed limitation in this case: the number of SIMD lanes must be a power of 2 not larger than 16. This kind of parallelism is by far more desirable than that achieved by the pipeline replication, since there is an opportunity to achieve coalescent memory accesses, due to the workitems belonging to the same workgroup. This is not the case with pipeline replication, as workitems executing in different pipelines are completely independent.

This technique is advantageous over pipeline replication as well in terms of chip area used, since control logic is not replicated, as in the case of pipeline replication. It is clear that this technique outperforms pipline replication by looking at figure 3.5. In this case, SIMD-4 seems to be best vectorisation factor, but this will greatly depend on the problem, as well as the work-group size.

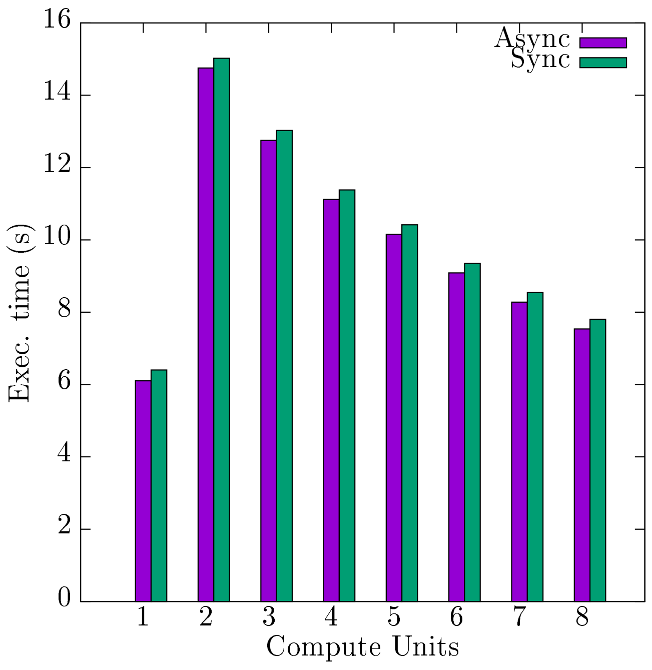
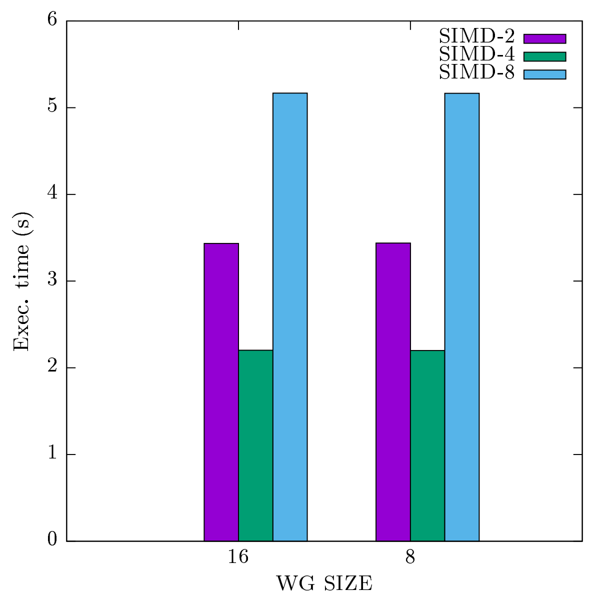


Figure 3.4: Performance of different number of replicas of CUs for unidimensional Jacobi

3.3 Local memory geometry: Matrix Transpose

In this section the behaviour of AOC when generating the geometry of local memories is going to be studied in order to determine whether manual intervention from the programmer is required in the general case and, as a consequence, it would be worth it giving support to the kernel parameters that allow their configuration. One of the simpler kernels yet conceptually rich in terms of memory access patterns is the matrix transpose. It will allow the discussion of the differences between GPUs and FPGAs and why the programmer must think differently when he switches between architectures. For that purpose, a CUDA implementation from the Nvidia SDK has been adapted to OpenCL [19], as shown in the code snippet 3.1. This kernel performs the matrix tranpose operation efficiently by leveraging local memory and an optimisation technique known as *tiling*, which consists of splitting a big array into smaller blocks to improve data locality. This way, it is possible to accelerate the operation by applying it over smaller blocks of the matrix in parallel, which are first loaded horizontally in the local memory *tile* and then read vertically and stored in the correspondent position in global memory. Since the tranpose operation occurs in local memory, every access to global memory is coalescent, thus achieving maximum bandwidth.

```
#define TILE_DIM 4

__attribute__((reqd_work_group_size(TILE_DIM, TILE_DIM, 1)))
            __attribute__((num_simd_work_items(TILE_DIM)))
__kernel void MatTranspose(__global float* restrict dest,
                           __global float* restrict src)
{
    __local float tile[TILE_DIM][TILE_DIM];

    int tx = get_local_id(0);
    int ty = get_local_id(1);
    int bx = get_group_id(0);
    int by = get_group_id(1);
    int x = bx * TILE_DIM + tx;
    int y = by * TILE_DIM + ty;
    int width = get_num_groups(0) * TILE_DIM;

    for(int j = 0; j < TILE_DIM; j += TILE_DIM) {
```



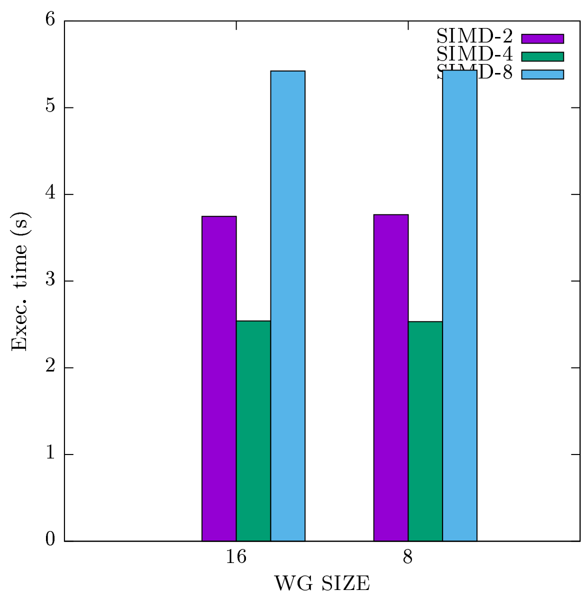


Figure 3.5: Performance of different number of SIMD lanes for asynchronous (left) and synchronous(right) execution of unidimensional Jacobi.

```
        tile[ty + j][tx] = src[(y + j) * width + x];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    x = by * TILE_DIM + tx;
    y = bx * TILE_DIM + ty;

    for(int j = 0; j < TILE_DIM; j += TILE_DIM) {
        dest[(y + j) * width + x] = tile[tx][ty + j];
    }
}
```

Listing 3.1: Matrix Transpose adapted from the CUDA implementation

Local memory (shared memory in the CUDA terminology) is organised either in 16 or 32 memory banks, depending on the GPU architecture. As memory banks are independent, groups of 16 threads (called warps) can access the local memory at the same time. Moreover, its SRAM technology allows random access without penalty. However, the programmer must be aware of the destination bank of every accessed memory position, since only one access is allowed at the same time. When there is more than one simultaneous request for a memory bank a conflict known as bank conflict occurs. In the matrix transpose example this happens because the local memory is written in a row major manner and read in column major manner. Figure 3.6 shows a simplified case with 5 memory banks. Note how when the second column is to be accessed a 5 bank conflict takes place. On the right, it is shown how the data layout changes to a diagonal distribution that prevents the problem. This work-around is not needed anymore when programming FPGAs, since the compiler takes care of the issue and synthesises appropriate interconnections. The novel programmer must be aware that applying this solution may result in a memory arbitration that would eventually cause poor performance.

The CUDA version of this code depends on vectorisation to achieve high performance. By default, AOC will synthesise the kernel as a pipelined circuit. In order to be able to apply the original idea in FPGAs it is necessary to apply pipeline vectorisation. It is possible to replicate the CUDA vectorisation by applying a SIMD factor of 16. It might not give the best performance, nonetheless, since the memory bus has a width of 64 bytes, which equals a coalescent request of 4 *floats*, i.e. it is expected that the

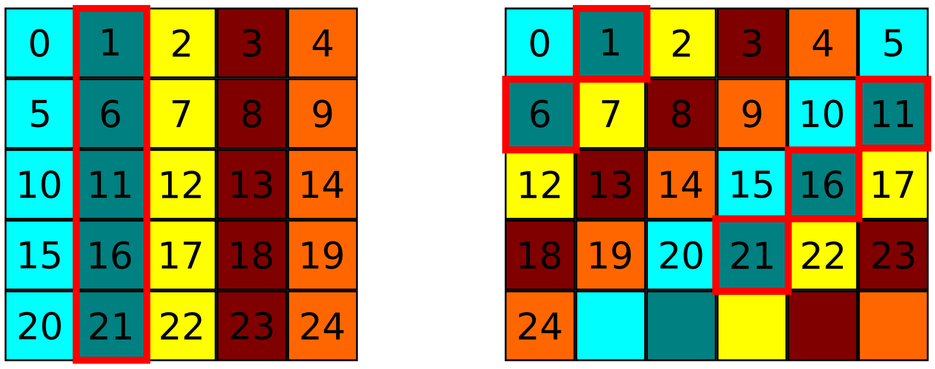


Figure 3.6: Local memory after being filled in a row major manner (left) and the same scenario when padding is applied (right)

maximum speedup is achieved at SIMD 4, whilst SIMD 16 might lead to some serious memory contention.

When applying SIMD in this kernel AOC gives the warning *Compiler Warning: Vectorized kernel contains loads/stores that cannot be vectorized. This might reduce performance.* alerting about a possible decrease in performance. This does not necessarily have to be the case. For further information, the compiler report can be consulted. There, the geometry of the local memory synthesised for *tile* is described, as can be seen in figure 3.7. The compiler opted for a single bank of memory with two replicates, thus allowing 4 read and 4 writes to take advantage of vectorisation. The 5 private copies of every replica enables the execution of 5 work-groups simultaneously, so no stage of the pipeline remains idle. The report also specifies that the LSUs perform 128 bits wide accesses. This confirms that SIMD has been successfully applied by the compiler, since *floats* are 32 bits wide.

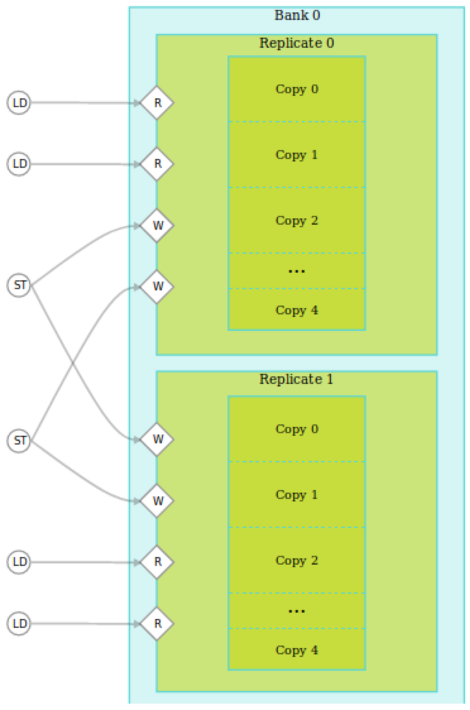


Figure 3.7: Local memory synthesised for *tile*.

3.4 Integration of FPGA as accelerator in Controllers

FPGAs can be programmed in the OpenCL programming model. This a standard for heterogeneous computing that allows the programmer to leverage the capabilities of different types of accelerators: CPUs, GPUs and FPGAs. The Trasgo research group had previously integrated GPUs, both Nvidia's and AMD's, in their model. There is a specific backend for Nvidia cards that was implemented in CUDA and a generic one implemented in OpenCL. The last one was used as a base to develop the backend for FPGAs. Thanks to the transparency for different architectures of OpenCL, integrating the FPGA for the execution of kernels was straightforward. Most of the work was dedicated towards exploiting specific characteristics of FPGAs that were not considered in the GPU backend. They are listed as follows:

3.4.1 Single work item kernels

Due to the inherent parallel nature of GPUs, kernels were executed in ND-Range fashion (refer to 2.2.3 for more information). Although this kernel launch model is supported by FPGAs, the single work item model adapts better to the pipeline architecture in the general case (there are some exceptions, such as the Matrix Power algorithm studied at 5). For this reason the Controller library is extended to support both models when targetting FPGAs that can be selected thanks to a parameter in the kernel characterisation, namely *NDRANGE* and *TASK*. Since there is no restriction on the execution model for a given problem (provided that the kernel code does not include explicit constraints) it is up to the programmer to choose the most adequate to achieve maximum performance.

3.4.2 Format of kernel name

It is now clear there are plenty of configurable parameters in FPGA kernels. Some of them have to be statically specified in the kernel code, as in the case of the pipeline configuration. The number of SIMD lines additionally requires to fix the workgroup dimensions. This is why the FPGA kernel binaries nomenclature had to be carefully chosen:

kname{**_SIMD-**N-XxYxZ}{**_CU-**M}{**_profiling**/**emu**}_version**.aocx**

Optional parts appear inside brackets in the kernel name definition. Parts in bold letter are mandatory, whilst the rest of the definition are variables:

- kname: name of the kernel as it will be called in the host code.
- N: number of SIMD lanes.
- X, Y, Z: dimensions X, Y and Z, respectively of workgroups.
- M: number of CU replicates.
- version: either Ctrl or Ref, standing for Controllers and native.

This simple solution allows the execution of instances of the same kernel with different compile-time parameterisations and prevents valuable information from being lost after the compilation process that can be useful for the programmer to use the kernel binary in other contexts different from that of the Controller library.

3.4.3 Expansion of kernel signature

The pipeline configuration is a decision with great impact on performance. For that reason, the kernel signature for FPGAs has been extended with a new parameter called *PIPELINE*. The purpose of this new parameter is to allow the programmer to set the number of SIMD lines and/or compute units and choose between the ND-Range or Task model of execution. This new directive must always appear in kernel signatures for FPGAs. It supports multiple parameters in order to enable sophisticated configurations of the pipeline. Caution is needed here, as some parameters clash between them because of their opposite semantics (as would happen in a raw native implementation). The possible parameters that *PIPELINE* can receive are:

- TASK: the kernel will be executed as a SWI (Single Work Item) kernel.
- NDRANGE: the kernel will be executed as an ND-Range kernel.
- SIMD(M, wg_x, wg_y, wg_z): the characterised kernel will execute in a pipeline with M lanes and workgroup dimensions (wg_x, wg_y, wg_z).
- CU(N): the kernel will execute in a pipeline replicated N times.

As the reader might have guessed *TASK* and *NDRANGE* cannot be used together in the same signature.

The offline compilation of kernels required by FPGAs imposes new restrictions on the manipulation of strings for the generation of the kernel code that is to be compiled, since many of those were performed at runtime with the OpenCL backend. To keep the changes of the *CTRL_KERNEL* primitive to the minimum the roles of the kernel arguments are added nested in the *PARAMS* macro and the kernel body is no longer a parameter but a block of code following the signature in a C function fashion.

```
CTRL_KERNEL(Mult, FPGA, PIPELINE( NDRANGE,
            SIMD( 4, 64, 64, 1) ),
            PARAMS( OUT, IN, IN, INVAL, INVAL ),
            KHitTile_float C, KHitTile_float A,
            KHitTile_float B,
            int A_width, int B_width)
```

Listing 3.2: Example of the new kernel signature

3.4.4 Kernel declaration

Controllers was initially thought as a unified host plus device programming model, leveraging the online compilation capability supported by the integrated architectures. However, FPGAs are an exception to this rule, since they can only operate with offline compilation. Kernels must be compiled separately from host code and the resulting kernel binaries must be provided to the host application. For this reason the code must be split into host and kernel files. In order to work around this limitation and preserve the transparency property of Controllers some effort was dedicated to hide it from the programmer.

Some code was moved to a header file included by both the host and the kernel file to minimise the changes to be made by the programmer to adapt his Controller code to work with FPGAs. These common parts were the definition of types used by Hitmap tiles, *Ctrl_NewType*, and any type alias made to allow the use of qualified types, such as *unsigned char*, in specialised *HitTiles*. In addition, the kernel execution mode (either ND-Range or Task) was added to the kernel characterisation as a parameter, which should match the one specified in *CTRL_KERNEL*.

Ctrl_NewType conveys different information depending on which file includes it. Hence, it is necessary for the preprocessor to distinguish between host code and kernel code in order to perform the appropriate macro expansions in every case. The less invasive way of achieving this result is the annotation of the kernel file with the empty macro *CTRL_FPGA_KERNEL_FILE*.

3.4.5 Managing incongruent grid sizes

Sometimes, computation is issued for a grid size that is not multiple of any of the block dimensions. As a result, the last block in that given dimension will have some of its work-items idle, since otherwise unallocated memory positions would be accessed. The OpenCL backend deals with this issue by checking that the index of the current work-item is between the grid borders specified in the structure Ctrl_Thread by the user with the function Ctrl_ThreadInit. This leads to a branch divergence i.e. work-items that meet the condition will perform the computation, whilst those that do not will return. Although this solution works for GPUs, FPGAs cannot cope with branch divergence in SIMD pipelines, since it is not synthesisable.

An alternative approach, which might lead to minor performance improvements, and can be extended to every architecture supported by Controllers with minor effort is the use of padding. As has been already mentioned, the problems with vectorisation come from checking the work-item indices belong to the index space, which prevents going out of bounds when accessing memory. The adopted solution comes through deleting the need for testing such condition. This is achieved by allocating enough memory for a big block size. The chosen size is 256 in every working dimension, since these are the default dimensions assumed by AOC for a SIMD kernel with a synchronisation point when they are not specified by the user.

The user will not perceive any alteration in the memory he requested, since the *HitShape* created will be preserved. To this end, the new memory size the backend will work with will be stored at the tile members *acumCard* and *origAcumCard[0]*. This way, some trash computations will be performed, as they will be out of bounds according to the dimensions chosen by the user, and they will not be visible because they exceed the boundaries specified in the requested shape initially.

3.4.6 Replacing pinned memory by aligned memory

In order for DMA transfers to take place in the CUDA programming model memory has to be pinned [18]. The term pinned memory refers to memory pages in the host address space that will not be moved to swap storage during the execution of the program. This way it is guaranteed that the page will be available when the DMA controller initiates a transference between host and device. In case the programmer does not explicitly allocate memory as pinned, the runtime will create a temporary pinned copy of the page to be transferred, affecting performance. On the other hand, memory is a scarce resource and pinning abuse may also lead to performance decrease. There is no way to guarantee the allocation of pinned memory in OpenCL, however, as the Nvidia OpenCL Best Practices Guide states it is possible to give a hint to the runtime to act this way. The process for host to device transfers, depicted in figure 3.9, is as follows (the opposite device to host is analogous):

- Create data buffer for transfers.
- Create auxiliary buffer to indicate OpenCL the corresponding mapped host memory will not be swapped out.
- Map auxiliary buffer to host memory.

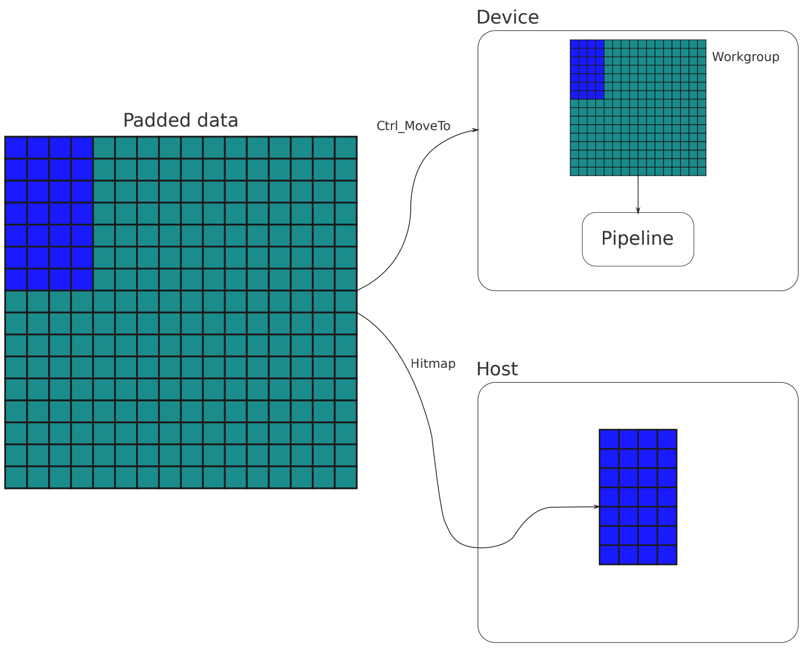


Figure 3.8: The figure shows a case in which the user allocates a 6×4 matrix. For the sake of simplicity and to keep the figure clear, the Task model of execution is supposed, hence, a single work-group is launched, and the default work-group size for padding has been reduced from 256x256 to 16x16. Although the user requests a 6×4 matrix, it is padded until the data fits into 16x16 work-groups. In this case a high percentage of the allocated memory is just due to padding because the matrix is extremely small. This will not happen in real applications. The device computes the kernel with the padded data, but the programmer only sees the data he requested, keeping this process transparent

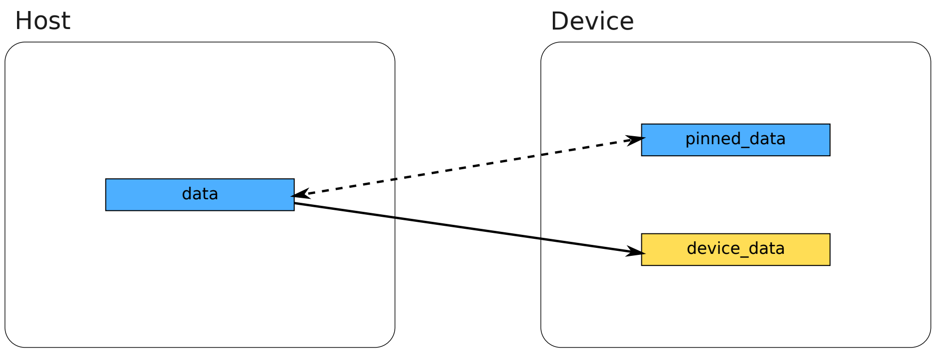


Figure 3.9: Pinned memory in OpenCL (Nvidia's implementation). The dashed arrow indicates mapped memory, whilst the continuous one stands for the direction of the memory transfer

This construct has no interest when programming for FPGAs, however, as it is not mentioned in any of the Intel guides nor in the scientific literature and none of the author tests point in the opposite direction. Rather, it just contributes to the occupancy of the global memory of the device.

Intel FPGAs, on the other hand, benefit from aligned memory to to achieve higher transfer rates thanks to DMA. The memory bus of these device is 64 bytes wide, i.e, data is transferred in that size bursts. Thus, memory buffers are aligned to 64 bytes in this solution to take advantage of this feature. More information on memory alignment in chapter 2.

3.5 Conclusions

This chapter was a summary of the configurable parameters of Intel FPGA SDK for OpenCL and showed a brief study that justifies the inclusion of support for vectorisation and compute unit replication in Controllers. Finally, the specification of the new backend was detailed.

Chapter 4

Implementation

This chapter introduces the following aspects:

- Description of the approaches to the final implementation of the FPGA backend of the Controllers model.
- Removal of initialisation operations from runtime code sections.
- Enabling profiling and emulation of kernels.

4.1 Naïve approach to the solution

The first milestone of the problem solved in this work was getting the Controllers model to execute FPGA kernels. Since FPGAs can be programmed in OpenCL, this was achieved by modifying the existing OpenCL backend for GPU. As a consequence, this was quite straightforward, although this solution just gave partial support to the Controllers instrumentation i.e. management of queues with asynchronous and synchronous policies and data transfers, but did not leverage the Hitmap library nor eased the seamless definition of kernels.

4.1.1 Changes made to the GPU backend

In this section the changes applied to the OpenCL backend in order to serve as a draft for this work are detailed. To start with, the OpenCL backend for GPUs was copied and every function, macro or variable referring to the previous backend nomenclature was renamed to prevent collisions between both backends when they are running at the same time to support GPUs and FPGAs.

Offline compilation

Before this work, Controllers dealt with kernels as strings in order to perform online compilation. FPGAs do not support this compilation mode and need to load a kernel binary compiled previously with a specific compiler (AOC - Altera Offline Compiler in the case of Intel FPGAs) in a process known as offline compilation. In order for the new FPGA backend to support this feature it was only necessary to read the precompiled kernel binary, whose path was hardcoded, and feed it to the function **clCreateProgramWithBinary** instead of **clCreateProgramWithSource**, which takes a kernel string as argument. The online compilation takes place when the function **clCreateKernel** is called. In the case of offline compilation it is turned into a dummy function that still has to be called to adjust to the OpenCL standard and only serves the purpose of returning an OpenCL kernel object built from the binary.

Constraints on passing structures

Controllers wraps data into a simplification of Hitmap tiles known as KHitTiles in order to allow data partitioning. This cannot be done straightforwardly, since the OpenCL standard states that structs with pointer members cannot be passed as kernel arguments (section 6.8.d) [16]. The OpenCL backend for GPU works around this by unfolding the tile arguments into coordinates and the pointer to data. The tile coordinates are saved into a KHitTile wrapper and the data is passed by reference. The Intel FPGA SDK for OpenCL has an extra limitation in old versions of the compiler on passing data structures: they cannot be passed by value. Instead, struct arguments must be converted into pointers to struct [22]. As host pointers are meaningless for the device, pointers to struct must be passed through an OpenCL buffer. All this manipulation was suspected to incur in some serious overhead and seemed too much effort for a naïve approximation of the solution which only looked for a functional implementation. Instead, the KHitTile struct was fully unfolded and every member was passed individually as a parameter. This implied kernels had to be adapted to receive all these dummy (recall they are intended to leverage the Hitmap library, which will not be used yet) arguments. Since this limitation is resolved in new versions of the compiler, there is no point in including support for this in the backend, so it was overlooked in the development of the definitive solution.

ND-Range kernels and Tasks support

The only difference in host code between launching an ND-Range kernel or a Task is calling the function **clEnqueueNDRangeKernel** or **clEnqueueTask**, respectively. Additionally, **clEnqueueNDRange** needs the grid and workgroup dimensions. Both functions calls were typed in the FPGA backend and every time a kernel of one kind was to be launched the call of the opposite kind was commented out.

4.1.2 Changes made to the experimentation code

The advantage of using Controllers over other heterogeneous programming models is highlighted by the minimal changes that have to be made in the experimentation host codes to launch kernels on FPGA. In fact, only the used device had to be changed to FPGA.

Some other changes had to be performed, but they are totally unrelated to the Controllers model, and are only due to the kernels being different from the GPU version. Although the GPU kernels can be executed on FPGA it is not recommended, since it is an entirely different architecture and the performance achieved will be poor in most cases. For this reason, sometimes the arguments passed to the FPGA kernels vary in number or type with respect their GPU counterparts, which has to be reflected in the kernel prototype.

In the case of the native version of experimentation host codes the offline compilation and ND-Range kernels and Tasks support explained in the previous point had to be added.

4.2 Approaches considered before definitive solution

It is clear that some of the decisions took in the naïve approximation can be considered bad practice and are not acceptable in a final product. It serves its exploratory purpose as a prototype well, nonetheless. During the development of this work several approaches were tried in order to satisfy the goal of integrating FPGAs in Controllers in an iterative manner. In this section every approach and the reasons why it was not the most appropriate alongside some thoughts on the knowledge gained from each of them, which influenced the following approach, are presented in this section.

4.2.1 First steps towards generalisation

The naïve approach requires programmer intervention in the FPGA backend to change the kernel used and specify its launching mode (ND-Range or Task). This is unacceptable and should be moved to the Controllers frontend.

Choosing the desired kernel to launch

Both the pipeline configuration and the device for execution (actual hardware or FPGA emulator), alongside profiling mode or normal mode are decisions that have to be made prior to the kernel compilation. For this reason, these are static features of the kernel binary and a different configuration of these parameters requires a new compilation. Given that kernel compilation is an expensive process that can take up to several hours it is advisory to save these binaries. There is no simple way of retrieving all this parameters from the compiled binary apart from consulting the HTML report generated during compilation, which is not easy to automate. A plausible solution which helps getting all the binaries organised is naming them according to the nomenclature explained in 3.4.2. It has also the advantage of allowing the mapping of the kernel characterisation detailed in 3.4.3 to the appropriate kernel binary by the manipulation of the string of the kernel path.

Specifying the kernel launching mode

Launching a kernel as ND-Range or as a Task not only implies how many workitems will execute it but prior work of adaptation of the whole kernel has had to be made in order to adapt to one of them. Although every kernel can be launched in both modes (unless an explicit restriction is expressed in the kernel file) this adaptation work is necessary to achieve high performance. Therefore, it seems reasonable to consider this decision part of the characterisation of the kernel, specifically of the configuration of the pipeline. The concrete parameters were explained in 3.4.3.

The annotation **__attribute__((max_global_work_dim(0)))** in the kernel file restricts its execution to Task mode and encourages AOC to try to optimise the pipeline in this direction. This has been exploited in Controllers by the creation of a series of new macros which will ultimately expand to the attribute conditioned to the definition of *CTRL_FPGA_KERNEL_FILE*. The mechanism used to achieve this has been the concatenation of the parameter passed to *KERNEL_PIPELINE* with the token *EXTRACT* with the operator ##. This way, when the concatenation result is *CTRL_EXTRACT_KERNEL_TASK*, a macro defined in the backend, the preprocessor will expand it to the given definition, which in this case is the said attribute. This mechanism has been used for the rest of the arguments.

4.2.2 First approach: Unified host+device programs

As has been discussed in the previous chapter the use of OpenCL on FPGAs requires host code and kernel code to be split in two different files, so that offline compilation is possible. This clashes with the transparency property of Controllers, i.e., the programmer faces the same programming interface regardless of the device he intends to use. The way to achieve this is by shifting the task of splitting the Controllers code from the programmer to another preprocessing step.

Lex

The tool chosen to achieve this result was Lex (specifically the GNU flavour Flex). Lex is a tool for generating scanners, also known as lexers: programs which recognise lexical patterns in text. It is based on the recognition of patterns by leveraging regular expressions. Unlike sed, Lex is designed to work over whole files. Moreover, it allows intermingling C code, which makes Lex a more versatile tool.

Structure of a kernel file

The code snippet 4.1 shows how a kernel file prepared for offline compilation should look. It is worth noticing every construct shown here is also necessary for the compilation of the host code. For this reason, the annotation *CTRL_FPGA_KERNEL_FILE* is necessary to allow the preprocessor to distinguish between the backend implementation targeted to the host and to the device.

```
#define CTRL_FPGA_KERNEL_FILE
#include "Ctrl.h"

Ctrl_NewType( float );

CTRL_KERNEL_CHAR(Mult, MANUAL, LOCAL_SIZE_0, LOCAL_SIZE_1, KERNEL_PIPELINE(KERNEL_SIMD(
     4, 64, 64, 1)));

CTRL_KERNEL_PROTO( Mult,
  1, FPGA, 5,
  OUT, HitTile_float, C,
  IN, HitTile_float, A,
  IN, HitTile_float, B,
  INVAL, int, A_width,
  INVAL, int, B_width
);

CTRL_KERNEL(Mult, FPGA, KHitTile_float C, KHitTile_float A, KHitTile_float B, int
    A_width, int B_width,
(
  ...
));
```

Listing 4.1: Structure of a kernel file after preprocessing step

Extraction of kernel constructs from unified code

The Lex program created for the extraction of kernel constructs from unified code is shown in code snippet 4.2. Specifically, it extracts *CTRL_KERNEL*, *Ctrl_NewType*, *CTRL_KERNEL_CHAR* and *CTRL_KERNEL_PROTO*. Since the input file is scanned sequentially three buffers have to be defined to allow the recognition of the constructs in no particular order, i.e, assuming they are indenpendent of each other. Some regular expresions, such as those referring to macro arguments or the kernel prototype, have been given an alias based on semantics to improve readability. Scanning the kernel body in *CTRL_KERNEL* and the primitive *CTRL_KERNEL_CHAR* was done by declaring two start conditions that drive the automaton to the states that deal with the problem of matching parentheses. The rest of the code is self explanatory.

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    #define CTRL_KERNEL_BUF 100000
    #define CTRL_KERNEL_PROTO_BUF 1000

    int unmatched_parens = 0;
    char ctrl_kernel_buf[CTRL_KERNEL_BUF];
    char ctrl_kernel_proto_buf[CTRL_KERNEL_PROTO_BUF];
    char ctrl_kernel_char_buf[CTRL_KERNEL_PROTO_BUF];
%}

%option noyywrap

spaces [\t\n ]
sep ;
arg [ A-Za-z0-9_*-]+{spaces}*
new_t Ctrl_NewType{spaces}*"("([^\)]|{spaces}])*")"{spaces}*{sep}*{spaces}*
k_char_1 CTRL_KERNEL_CHAR{spaces}*"("({arg},{spaces}*)+KERNEL_PIPELINE"("
k_proto CTRL_KERNEL_PROTO{spaces}*"("(({arg}|,){spaces}*)+")"{spaces}*{sep}*{spaces}*
ctrl_k_signature CTRL_KERNEL{spaces}*"("{spaces}*({arg},)+[\t\n ]"("


/* extracts argument inside parenthesis */
%x kernel_body k_char_args
%%
{new_t} {
    strcat(ctrl_kernel_char_buf, yytext);
}
{k_char_1} {
    strcat(ctrl_kernel_char_buf, yytext);
    unmatched_parens = 1;
    BEGIN k_char_args;
}
{k_proto} {
    strcat(ctrl_kernel_proto_buf, yytext);
}
{ctrl_k_signature} {
    strcat(ctrl_kernel_buf, yytext);
    unmatched_parens = 1;
    BEGIN kernel_body;
}


.|\n ;

<kernel_body>{spaces}*[^\(^\)]+"(" {
    unmatched_parens++;
    strcat(ctrl_kernel_buf, yytext);
}
<kernel_body>[^\(^\)]*")"{spaces}*{sep}*{spaces}* {
    strcat(ctrl_kernel_buf, yytext);

    if (!unmatched_parens)
        BEGIN INITIAL;

    unmatched_parens--;
}
<kernel_body>(.|\n) ;

<k_char_args>{spaces}*[^\(^\)]+"(" {
    unmatched_parens++;
    strcat(ctrl_kernel_char_buf, yytext);
}
<k_char_args>[^\(^\)]*")"{spaces}*{sep}*{spaces}* {
    strcat(ctrl_kernel_char_buf, yytext);

    if (!unmatched_parens)
        BEGIN INITIAL;

    unmatched_parens--;
}
<k_char_args>(.|\n) ;


<<EOF>>   { return 0 ; }

%%
int main() {
    yylex();

    printf("#define CTRL_FPGA_KERNEL_FILE\n");
    printf("#include \"Ctrl.h\"\n\n");
    printf("%s\n", ctrl_kernel_char_buf);
    printf("%s\n", ctrl_kernel_proto_buf);
    printf("%s\n", ctrl_kernel_buf);
}
```

Listing 4.2: Lex program for the extraction of kernel constructs from unified code

Reasons for rejection

This approach contributes to a uniform view of the Controller model by avoiding the separation of host and kernel code, thus keeping transparency. Otherwise the programmer has to take into account that FPGAs are a special case and split the program, indicating which file contains the kernel code with the *CTRL_FPGA_KERNEL_FILE* annotation. However, this solution relies on Lex, and this implies that Controllers will now have a dependency on an external tool. As stated in 2.2.4 one of the main advantages over other models is it is compiler agnostic, and Lex could be considered a source-to-source compiler in this case which would remove this key feature from Controllers.

4.2.3 Second approach: First steps towards compiler agnostic compilation

Although the single file property is a desirable feature, it comes at the cost of the dependency on the external tools sed and Lex. This would impose that the programmer using Controllers had these programs installed on his machine and the model would stop being compiler agnostic, which is one of its key advantages over other models. Neither the single file property nor relieving the programmer from writing the annotation *CTRL_FPGA_KERNEL_FILE* outbalance the benefits of keeping Controllers independent of the compiler used, so the definitive solution will rely on the preprocessor to achieve this goal.

Kernel arrays are wrapped into *KHitTile* structures, which track the shape coordinates and keep a pointer to its data. This pointer has to be meaningful for the device i.e. it cannot point to the host memory space, so the data must be copied to memory accessible by the device. This is easily achieved in CUDA by declaring the data pointer as *void ** and making it point to a device buffer. The entire *KHitTile* structure will be passed by value to the kernel device function. This is not possible due to the restriction that forbids passing structures with pointer members by value. The only possible workaround for this, that was previously implemented in the OpenCL backend, is the extraction of the data pointer from the *KHitTile* structure. In order to keep the body of kernels the same for the user regardless of the architecture, the structure *KHitTile_wrapper* is created, which holds the coordinates in the corresponding *KHitTile*. The signature of the kernel changes and now a *KHitTile_wrapper* and the data pointer are passed instead of the whole *KHitTile*. The latter will be assembled inside the kernel and named as the original parameter specified by the user, avoiding this way any modification in the code that accesses global memory.

The approach taken in the OpenCL backend has the major pitfall of depending heavily on compile-time string manipulation, which hinders its reuse in the FPGA backend. Specifically, the data pointer type is extracted from the *HitTile_<type>* passed to the kernel prototype macro. This is possible by leveraging a generic macro that expands to a variable (*raw_ktile_KHitTile_##type*) that holds the *<type>* in *HitTile_<type>* as a string which can be concatenated to the rest of the kernel argument later, as shown in the code snippet 4.3. Instead, the data pointer might be *void ** typed. These kind of pointers must be casted before being dereferenced. The assembly of the *KHitTile_<type>* would perform an implicit cast of the data pointer to *<type>* and the problem would be solved. This requires pasting the *KHitTile_<type>* assembler code to the kernel code using the concatenation operator *#*. This would work for unoptimised codes that do not make use of the *#pragma* directive, as embedding it in a macro is not supported by the Altera Offline Compiler. As a side note, the option of preprocessing the kernel file with CPP and feeding the result to AOC was also considered as a workaround for this limitation and it was found out that although it supports embedding pragmas in macros, it moves the directive to the beginning of the expanded code. This is an undocumented behaviour that can be amended by replacing *#pragma* by *__Pragma("GCC *")* in C programs [2]. However, this substitute will not be recognised by AOC, so *CTRL_KERNEL* must be modified so the kernel is not passed as a macro parameter anymore. Instead, the original *CTRL_KERNEL* from the first Controller versions is retrieved, i.e, the argument is removed and a code block (delimited by brackets *{* and *}*) containing the kernel is attached to the macro, resembling a function definition.

```c
#define hit_ktileNewType( type ) \
    typedef struct { \
        type  * data; \
        int origAcumCard[4]; \
        int   card[3]; \
    } KHitTile_##type;\
    char *raw_ktile_KHitTile_##type = CTRL_KERNEL_STRINGIFY( type ); \
    ...
    bool raw_added_ktile_KHitTile_##type = false;
```

Listing 4.3: Retrieving KTile type in OpenCL. This macro is called from *Ctrl_NewType*

A clear disadvantage of having the kernel code surrounded by brackets is that there is no way for the preprocessor to paste code next to it, so the *KHitTile_<type>* structures cannot be assembled in the device during the kernel execution. Given that the Hitmap macro *hit* used in the Controllers kernels to access the data relies on these structures, they must be assembled anywhere else. Recall that *KHitTile_<type>* structures cannot be passed by value, but there is no restriction on passing a pointer to them. Therefore, the strategy followed is the creation of a new kernel that receives the *KHitTile_wrapper* and the data and saves the assembled *KHitTile_<type>* structure in global memory. Finally, the pointer to its memory position is passed to the kernel. Despite looking like a manipulation close to a naïve function refactoring, it requires several modifications to Controllers.

Kernel initialisation

The new ktile assembler kernel is injected in the macro *Ctrl_NewType*, given that it must be generic and dependent on the ktile type. We will call this kernel pre-kernel from now on. It is clear that the pre-kernel must be called before the kernel in order to have the ktiles prepared for its use. Thus, every initialisation step prior to OpenCL kernels launch must have been completed before the kernel is to be enqueued. At the moment, this is achieved the first time the kernel is called with the macro *Ctrl_Launch*. Not only does this prevent launching kernels in earlier stages of the Controllers program, but also is quite inefficient. Notice that this involves the creation of every kernel object that will be used by the program. In the case of FPGAs this is even worse, as kernels are stored in binaries that have to be retrieved from disk, severely affecting the time measurements in experimentation (unless they are actually read from a ramdisk, but such specific infrastructure should not be expected from the user).

```
__kernel void CTRL_KERNEL_POPULATE_KHITTILE_##type(KHitTile_fpga_wrapper ktile_wrapper,
    __global void* restrict data, \
      __global KHitTile_##type* restrict ktile) { \
      ktile->origAcumCard[0] = ktile_wrapper.origAcumCard[0]; \
      ktile->origAcumCard[1] = ktile_wrapper.origAcumCard[1]; \
      ktile->origAcumCard[2] = ktile_wrapper.origAcumCard[2]; \
      ktile->origAcumCard[3] = ktile_wrapper.origAcumCard[3]; \
      ktile->card[0] = ktile_wrapper.card[0]; \
      ktile->card[1] = ktile_wrapper.card[1]; \
      ktile->card[2] = ktile_wrapper.card[2]; \
      ktile->data = data; \
    }
```

Listing 4.4: New ktile assembler kernel

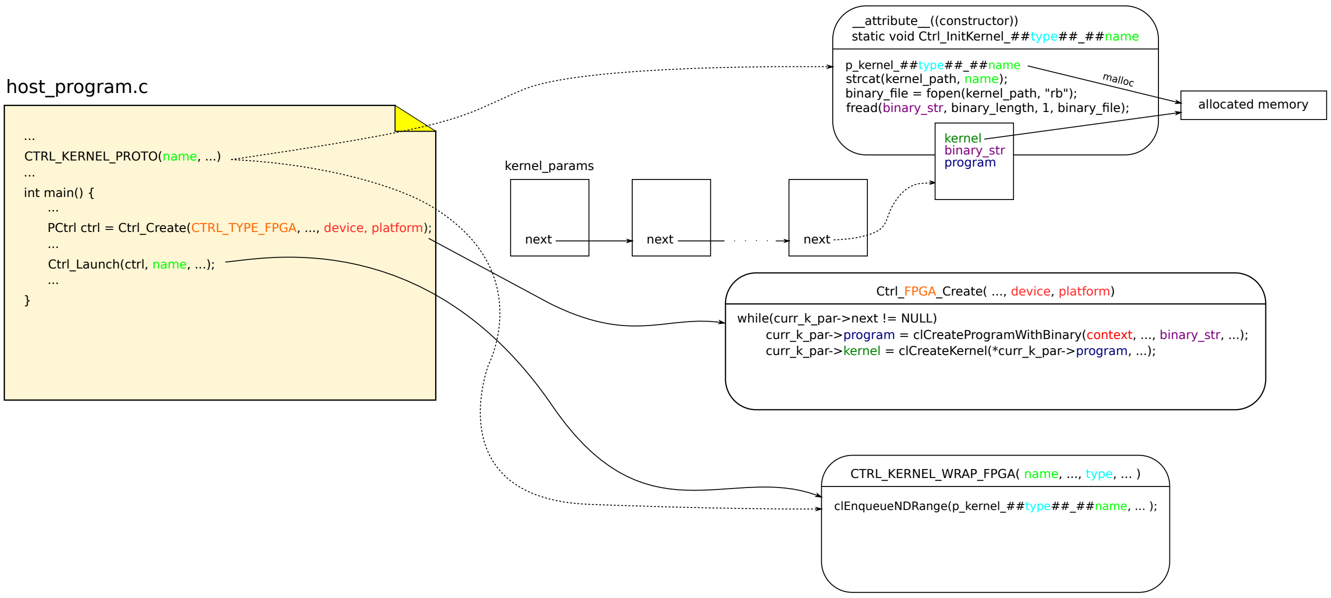
All in all, this initialisation step must be moved before the first kernel needs it to have completed. As the pre-kernel will be launched for every tile created by the user when he summons the macro *Ctrl_Alloc* and both the name of the kernel and the chosen architecture are needed (recall, for example, that the kernel pointer is of the form *p_kernel_<type>_<name>*) it seems reasonable to have it in the macro *CTRL_KERNEL_PROTO*. Two subtleties must be considered here: the code expanded from this macro will be placed before the main function, i.e, no code can be executed except from definitions and declarations and some information is needed to build the kernel that is not available until the Controller is created (specifically, platform and device). The first problem is solved by wrapping the initialisation code in a constructor function [1]. The second requires a more sophisticated solution. First a new structure must be created and declared as global. It will work as a linked list and every element will contain the information needed for the initialisation, as shown in the code snippet 4.5. For every new kernel prototype a new element is added to the linked list and each pointer member is made to point to the memory pointed by the OpenCL objects created in the initialisation needed for the kernel creation. Notice how the memory pointed by *p_kernel_<type>_<name> ** would not be accessible from anywhere else than the kernel prototype scope as the parameters *<type>* and *<name>* are no longer available. This way, the creation of the kernel can now take place when the Controller is created and the chosen device and platform objects are available. This modifications are depicted in figure 4.1. Of course, the initialisation code must be split between the kernel prototype and the Controller creation function (*Ctrl_FPGA_Create*). The keen reader might have noticed that this pattern has already been used to pass the initialised ktiles to the kernel from the pre-kernel.

```
struct prekernel_params {
  cl_kernel *prekernel;
```

[1]The typical C program has *main* as its entry point and every function has to be called from it, meaning that no code can be executed before *main* is called or after it has returned. Sometimes it desirable to execute some initialisation code so that certain resources are available when the program starts running. For this situations, *constructors* or *initialisation routines* can be leveraged. When a function is annotated as constructor the compiler stores it in an array in a special section, so that they can be called in order before *main* starts.

```c
  const char *prekernel_name;

  struct prekernel_params *next;
} prekernel_params;

struct kernel_params {
  cl_kernel *kernel;
  cl_program *program;
  const char *kernel_name;
  unsigned char *binary_str;
  size_t binary_length;

  struct kernel_params *next;
} kernel_params;

struct tile_params {
  size_t ktile_size;
  cl_kernel *p_prekernel;
};
```

Listing 4.5: New linked list of initialisation parameters



```
...
CTRL_KERNEL_PROTO(name, ...)
...
int main() {
    ...
    PCtrl ctrl = Ctrl_Create(CTRL_TYPE_FPGA, ..., device, platform);
    ...
    Ctrl_Launch(ctrl, name, ...);
    ...
}
```

```
__attribute__((constructor))
static void Ctrl_InitKernel_##type##_##name

p_kernel_##type##_##name
strcat(kernel_path, name);
binary_file = fopen(kernel_path, "rb");
fread(binary_str, binary_length, 1, binary_file);
```

```
Ctrl_FPGA_Create( ..., device, platform)

while(curr_k_par->next != NULL)
    curr_k_par->program = clCreateProgramWithBinary(context, ..., binary_str, ...);
    curr_k_par->kernel = clCreateKernel(*curr_k_par->program, ...);
```

```
CTRL_KERNEL_WRAP_FPGA( name, ..., type, ... )

clEnqueueNDRange(p_kernel_##type##_##name, ... );
```

Figure 4.1: Design modification that moves initialisation operations before the execution measure region. The rounded blocks represent either functions (lowercase) or macros (uppercase) with a simplification of the concerned code. Dependencies are shown in colours. The execution of every function/macro happens from top to bottom. Dashed arrows from the host program represent the preprocessor expansions, whilst the solid ones show the execution of code thanks to the evaluation of a task in the Controller queue.

Tile allocation

The assembly of the ktiles needs to be performed when the tiles are allocated, as that is the moment when the *HitTile* structures are built. Recall that in order to allocate the buffer that will store the ktile it is necessary to have its size at hand. This will depend on its <*type*> and this information is not available in the task evaluation functions that create the tiles. This functions must be kept unaltered, since they are part of the runtime, hence independent of the backends, so the size must be passed other way. The pre-kernel function needed for every given tile also depends on the the same parameter, so the pointer to the corresponding kernel object must also be passed. A brief explanation on how tiles are managed might be valuable to understand the adopted solution. As has already been mentioned, tiles are represented by the *HitTile* structure, which keeps track of coordinates and other useful information to ease complex memory access patterns. In order to manage both host and device copies of the tile and perform data movements according to different policies there is a need for more information. HitTile was designed to support an extension via its member *void *ext*. In Controllers it is leveraged by attaching a tile specialisation that contains all the extra information detailed. The object oriented approach taken in the development of Controllers allows to adapt this tile specialisation named *Ctrl_<type>_Tile* to every architecture. Thus , three new members are added to *Ctrl_FPGA_Tile*:

1. *cl_mem ktile*: device buffer that will store the *KHitTile_<type>* structure once it has been populated by the pre-kernel.
2. *size_t ktile_size*: size of the *KHitTile_<type>* that will be stored in the buffer *ktile*.
3. *cl_kernel *populate_ktile_kernel*: *<type>* specialised kernel that will assemble the ktile.

The void pointer member *ext* of *HitTile* can be used to pass the size of the ktile and the pointer to the pre-kernel, that will be packed in a structure , from *Ctrl_Alloc_<type>*. Once inside the creation of tiles function the *ext* member is needed to attach the specialised tile, so the members inside the struct are saved into the corresponding ones in *Ctrl_FPGA_Tile*. After that, the ktile buffer is created in the function that evaluates the tile allocation task making use of the new information and the pre-kernel is launched.

Parameters passing

Kernel launching is managed in Controllers by pushing a task in the queue associated to the controller. This way the task can be evaluated independently from the moment the user calls the kernels and can be reordered either to guarantee data consistency or to increase performance. Therefore, the parameters passed must be stored in a list. The task keeps a pointer to its head and a list of the sizes of every argument to ensure they can be accessed afterwards. Initially, kernels were supposed to receive ktiles by value, regardless of the architecture, so the sizes stored were that of ktiles of the corresponding type. This is not the case anymore, since the arguments of FPGA kernels are pointers to that ktiles. It might seem the OpenCL kernels also broke the rule, since they received a ktile wrapper and a pointer to data. However they were generated by splitting the original parameters. This requires adapting the list generation macros to store ktile buffers and the size list to store ktile pointers buffers. This macro will be conditionally expanded as explained for FPGAs for the moment, although it will be extended to the binary based OpenCL backend, when the migration takes place.

```
#define CTRL_KERNEL_KTILE_STORE_IN(list, type, name)     \
    *((cl_mem *)(list)) = ((Ctrl_FPGA_Tile*)(name->ext))->ktile; \

#define CTRL_KERNEL_KTILE_STORE_IO(list, type, name)     \
    *((cl_mem *)(list)) = ((Ctrl_FPGA_Tile*)(name->ext))->ktile; \

#define CTRL_KERNEL_KTILE_STORE_OUT(list, type, name)    \
    *((cl_mem *)(list)) = ((Ctrl_FPGA_Tile*)(name->ext))->ktile; \
```

Listing 4.6: Parameters passing

Performance penalty induced by AOC

Once this solution was implemented it was discovered via experimentation with the Sobel kernel that it suffered from huge overhead, with execution times of around 20 seconds for a single frame. The profiler results showed that the memory bandwidth achieved by this kernel was 179.7 MB/s (1.61% of efficiency). Since the only change in the kernel was the replacement of ktiles by pointer to ktiles a simple experiment was conducted to study the behaviour of the offline compiler with two minimal kernels that reproduced this problematic scenario. The code in 4.7 has two kernels that are functionally equal and are expected to be synthesised the same way. Note that the only formal difference is how arguments are passed. In the case of *struct_with_pointer* the struct with pointer member datatype *swp* is passed and a local copy is instanced inside the kernel, whilst in *struct_without_pointer* a struct without a pointer member is passed instead alongside a pointer and both arguments are assembled into a *swp*.

```
typedef struct {
    int x;
    int y;
    __global int *restrict data;
} swp;

typedef struct {
    int x;
    int y;
} swop;

__kernel void struct_with_pointer(__global swp *restrict p_in, __global swp *restrict
    p_out) {
    swp in = *p_in;
    swp out = *p_out;

    for(int i = 0; i < 10; i++)
        out.data[i] = in.data[i] + 3;
}

__kernel void struct_without_pointer(swop in_coords, __global int *restrict data_in,
                                     swop out_coords, __global int *restrict data_out) {
    swp in = {
        .x = in_coords.x,
        .y = in_coords.y,
        .data = data_in
    };

    swp out = {
        .x = out_coords.x,
        .y = out_coords.y,
        .data = data_out
    };

    for(int i = 0; i < 10; i++)
        out.data[i] = in.data[i] + 3;
}
```

Listing 4.7: Simple experiment to check the behaviour of AOC with struct and pointer to struct arguments

It is clear that AOC is unable to recognise that both kernels are equal and has to generate more complex arbitration logic to ensure proper functionality. This leads to slower LSUs, as appears in table 4.1. Attending just to the sum of the latencies of the LSUs shown in the table the *struct_with_pointer* kernel will execute 30 times slower. This finding about the behaviour was confirmed as the explanation for such a poor performance in Sobel by comparing the report generated for the Controllers version with that of the native.

In conclusion, this solution has to be discarded due to an undocumented behaviour of the offline compiler. Nevertheless, it is solid enough as to be considered for future versions of Controllers in case Intel amends its compiler decisions for this scenario.

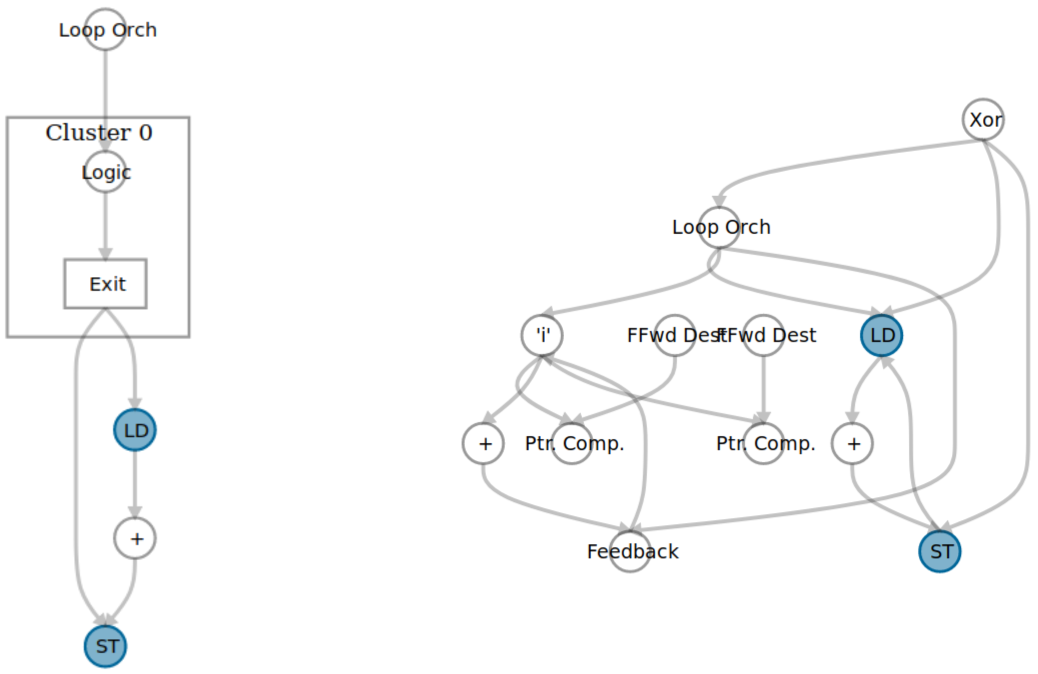


Figure 4.2: Arbitration logic in *struct_without_pointer* (left) and *struct_with_pointer* kernels (right)

	struct_with_pointer	struct_without_pointer
LD	129	26
ST	3	2

Table 4.1: Load store units latencies in the experiment kernels

4.3 Definitive approach: returning to the ktile wrapper

As it has already been discussed, there is a performance penalty on using pointers to struct as kernel arguments. Given its magnitude there is no other workaround for passing the ktiles to the kernel than returning to the approach taken in the OpenCL backend. It could be easily adapted to FPGA by typing data pointers as *void ** (Recall the only information referring to the datatype of the data pointer is the HitTile type, which is of the form *HitTile_ <type>*) had it not been because this is done in *CTRL_KERNEL_PROTO*, which, as every C statement, has to be followed by *;*. Since the kernel signature is followed by the kernel body and nothing can stay in between, the *;* would have to be removed. It could be argued that Controllers would stop being a C based model, so this is not an acceptable workaround. Moreover, the kernel prototype was initially thought as an independent statement and generating the kernel signature from it would keep it fixed before *CTRL_KERNEL*. Instead, it will be expanded from *CTRL_KERNEL*. The same applies for kernel attributes, that were expanded from *CTRL_KERNEL_CHAR* in the unified version.

4.3.1 Expanding the kernel signature from *CTRL_KERNEL*

In order to generate the kernel signature it is necessary to have the roles of each argument, since ktiles have to be separated in the ktile wrapper and the data pointer and INVALs must be kept as they are. This information is not available in *CTRL_KERNEL*, but in *CTRL_KERNEL_PROTO*. It has to be duplicated in order to make this distinction between arguments. Take into account that *CTRL_KERNEL* is supposed to be seen as the kernel signature for the user of Controllers and the roles are just qualifiers to the arguments. In order to minimise the impact of including them here they will be surrounded by the macro *CTRL_KERNEL_FPGA_ROLES*.

The name of the arguments cannot be extracted leveraging the preprocessor, since it is preceded by its type and a white space. The concatenation operator cannot alter any of the characters of the token, but it is able to attach text to its sides. This allows the modification of the type and the name of the argument. On top of that, it is possible to assign an alias to a pointer with *typedef*. With these tools, the strategy is as shown in figure 4.3: now there will be a ktile wrapper structure for every type instantiated in the host code, so that it can be instantiated by attaching *fpga_wrapper_* to the beginning of the token. In order to type the data pointer as *void ** this datatype, alongside the keyword restrict, are aliased with *typedef* in the creation of ktiles of a given type, as well. A token will be concatenated to them, *_wrapper* and *_data*, respectively, in order to be distinguishable from the rest of the arguments.

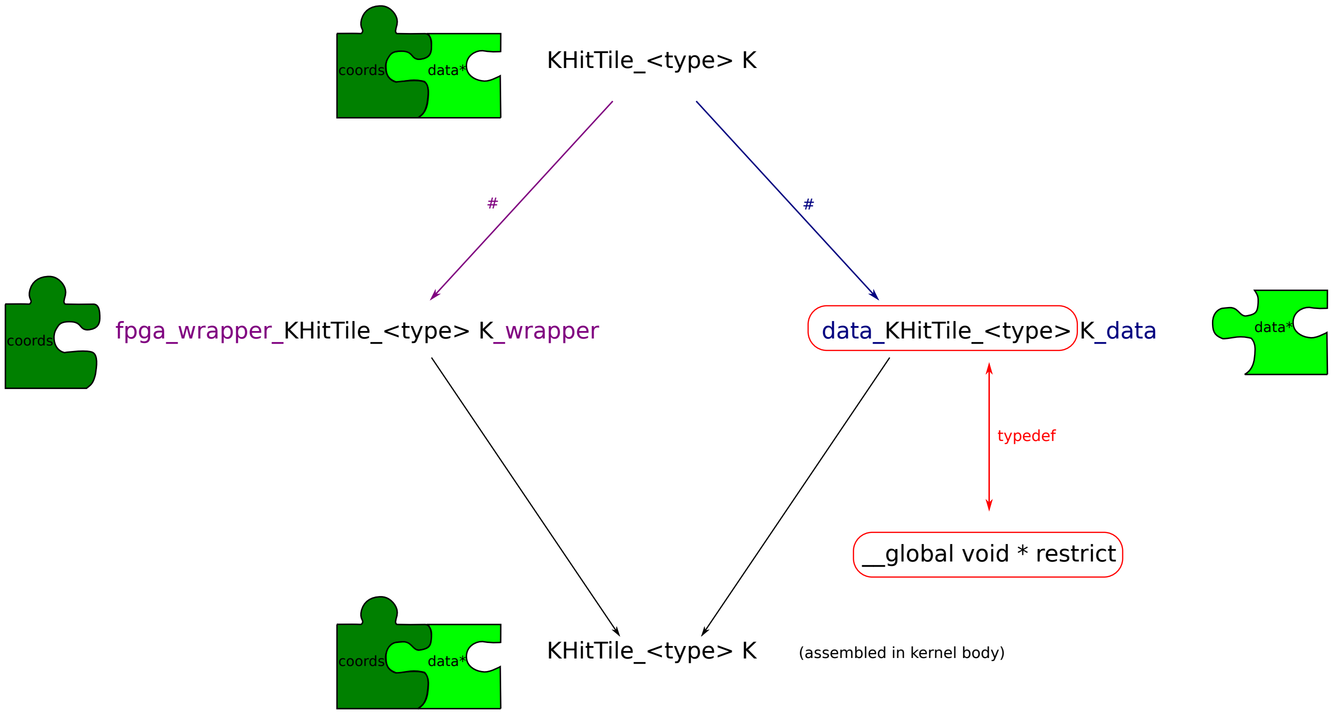


Figure 4.3: The concatenation operator and typedef are leveraged to generate the function signature from *CTRL_KERNEL*.

Special attention should be paid to the semantics of the address qualifier for the definition of the pointer alias. Defining the alias as *typedef void* restrict data_KHitTile_<type>* will result in the offline compiler giving the error *parameter may not be qualified with an address space*. In order to understand what is going on it is necessary to analyse the steps taken by the compiler. Since no address space is specified, the pointer points to the private address space. Hence *__global data_KHitTile_<type>* is indicating that the pointer itself resides in the global address space, so the compiler is understanding the opposite of what we want to convey: the pointer should reside in the private address space and should be pointing to the global address space. For that, the solution comes through including the address qualifier in the definition of the alias, so the compiler understands it refers to the memory pointed by the pointer, as in figure 4.3.

```
// Host code file
CTRL_KERNEL_CHAR(Mult, MANUAL, NDRANGE, LOCAL_SIZE_0, LOCAL_SIZE_1);

CTRL_KERNEL_PROTO( Mult,
  1, FPGA, 5,
  OUT, HitTile_float, C,
  IN, HitTile_float, A,
  IN, HitTile_float, B,
  INVAL, int, A_width,
  INVAL, int, B_width
);

// Kernel code file
CTRL_KERNEL(Mult, FPGA, PIPELINE( NDRANGE, SIMD( 4, 64, 64, 1) ), PARAMS( OUT, IN, IN,
    INVAL, INVAL ),
            KHitTile_float C, KHitTile_float A, KHitTile_float B, int A_width, int
    B_width)
{
    CTRL_INIT_KTILE(A, float);
    CTRL_INIT_KTILE(B, float);
    CTRL_INIT_KTILE(C, float);
    // Kernel body
}
```

Listing 4.8: Example of the definitive version of the Controllers interface modified macros

Expanding the kernel signature from *CTRL_KERNEL*, an alternative approach

Since the names of the arguments must be available for the generation of the kernel signature and leveraging the preprocessor to hide this requirement from the user by the last approach was unsuccessful, there is no other option than requesting the programmer to do so. To serve this purpose, the macro *KTILE(type, name)* is created, which only takes the type of the data pointer and the name of the argument as parameters. The roles are not needed anymore, since they were just used to differentiate between invals and ktiles and this is solved by the use of the new macro. The pipeline specification is also moved from *CTRL_KERNEL_CHAR* to *CTRL_KERNEL*, but the launching mode has to be preserved in both macros. It will be used to determine the launching model in the kernel characterisation macro and to generate the associated attributes in *CTRL_KERNEL*. A code excerpt involving the modified macros is given in 4.9, where it is clearly stated which belong to either the host or the kernel file.

```
// Host code file
CTRL_KERNEL_CHAR(Mult, MANUAL, NDRANGE, LOCAL_SIZE_0, LOCAL_SIZE_1);

CTRL_KERNEL_PROTO( Mult,
  1, FPGA, 5,
  OUT, HitTile_float, C,
  IN, HitTile_float, A,
  IN, HitTile_float, B,
  INVAL, int, A_width,
  INVAL, int, B_width
);

// Kernel code file
CTRL_KERNEL(Mult, FPGA, PIPELINE( NDRANGE, SIMD( 4, 64, 64, 1) ), KTILE(float, C),
    KTILE(float, A), KTILE(float, B), int A_width, int B_width)
{
    CTRL_INIT_KTILE(A, float);
    CTRL_INIT_KTILE(B, float);
    CTRL_INIT_KTILE(C, float);
    // Kernel body
}
```

Listing 4.9: Example of an alternative version of the Controllers interface modified macros

4.3.2 Profiling and emulation

Thanks to the kernel name format described in 3.4.2 the user is able to perform profiling or run a kernel on an emulated device just by enabling the corresponding option in the CMake file that compiles the Controllers project. Since this is a decision that conceptually affects how a kernels is executed but not its functionality or its eventual mapping on the device it seems more appropriate to keep these options outside the Controllers code.

To enable profiling the option *FPGA_PROFILING* must be set, whilst for emulation the same must be done with the option *FPGA_EMULATION*. As they are incompatible, when they are activated at the same time emulation will be chosen by default. What is happening behind the scenes is that the kernel path string is modified by inserting the word **profiling** or **emu**, depending on the enabled option, as specified by the kernel name format.

4.4 Conclusions

This chapter described the solution developed for the integration of FPGAs in the Controllers library. It enhances programmability, while allowing transparent dependency and data management and asynchronous execution, which leads to performance gains thanks to to the ability to overlap data transfers and computation. This was achieved at the cost of minor changes to the user-facing interface of Controllers when programming FPGAs in the definitive approach. They could be minimised even more in the future thanks to the second approach, which does not work at the moment due to the current behaviour of the Intel compiler. All the discarded approaches are functional, so the first approach could be leveraged in the future to remove any divergence the programmer can perceive in Controllers due to the use of this backend, provided the independence from external tools is not a concern anymore.

The initalisation operations related to kernels launching were removed from the runtime, as this became a need for the second approach. This might, in addition, lead to a potential improvement in time measurements that compare Controllers with other reference models.

Finally, profiling and emulation were eased for the programmer by requiring just the activation of a flag.

Chapter 5

Experimentation

This chapter introduces the following aspects:

- The objectives of the experimentation performed.
- The chosen case studies:
  - Hotspot
  - Matrix Power
  - Sobel Filter
- Performance study that validates our proposal.
- Study of software metrics on the generated code.
- The results obtained from the experimentation and the conclusions reached.

5.1 Objectives of experimentation

This section introduces the objectives of the experimentation for the proposed model. This objectives are as follows:

1. The performance improvement due to overlap depends on the workload of the communication and computation tasks and the dependencies between them.
2. The solution proposed over the Controllers library offers a code implementation with less complexity over other existing technologies.
3. The overhead of this abstraction, due to the implemented prototype and the internal control structures, has a minimal effect over performance in parallel executions compared to solutions over other exising technologies.

5.2 Case study: Hotspot

The base program is an adaptation of that of the Rodinia suite of benchmarks by Zohouri et al. [50, 51, 35, 53]. It computes the stability point of the Poisson's Partial Differential Equation (PDE) for heat diffusion. It uses a Jacobi iterative method on a 2-dimensional discrete space. It is a 4-point stencil program that executes a fixed number of time iterations. The kernel exploits the shift register pattern for both spatial and temporal locality (refer to 2.1.6 for further information). The program is tested with 400 iterations. The result matrix is transferred to the host after each kernel launching operation, saving it in a different host buffer using a host-task. Thus, the results could be used to check partial results or create an animation of the computation evolution. Matrices with input sizes from 1024×1024 to 4096×4096 are considered in order to achieve a memory-bound scenario.

Algorithm 1 Hotspot

```
 1: function Copy($A, B$)
 2:     Data: x, y: column and row indices, respectively
 3:     $B[x][y] \leftarrow A[x][y]$
 4: end function

 5: function HotspotIteration($P, A, B$)
 6:     Data: x, y: column and row indices, respectively
 7:     $B[y][x] \leftarrow P[y][x] + (A[y-1][x] + A[y+1][x] - 2A[y][x])/R_x +$
                    $(A[y][x-1] + A[y][x+1] - 2A[y][x])/R_y +$
                    $(T - A[y][x])/R_z$
 8: end function

 9: function main
10:     Data: n: number of iterations, P: power matrix, A: original temp. matrix, B: temp. matrix copy
11:     for $i \leftarrow 1, n$ do
12:         HotspotIteration(P, A, B)
13:         MoveFrom(B)
14:         Copy(B,A)
15:     end for
16: end function
```

5.3 Case study: Matrix Power

This program is an evolution of the *2mm* and *3mm* programs in the PolyBench Benchmarks [36] to generate a chain of matrix multiplications of arbitrary length. It computes the normalisation of the matrices $C_i = A^i : i \in [1 : n]$. It iteratively computes in the device the multiplication of the original matrix by the partial result of the previous step: $c^k = C^{k-1} \times A : k \in [1 : n]$ where $C^0 = A$. The kernel to multiply matrices is obtained from the Intel FPGA Support Resources. This optimised kernel uses local memory, loop unrolling and SIMD to take advantage of the FPGA resources. Each partial result C^i is transferred to the host. A host-task computes the normalistion of the matrix and saves it in another buffer. The matrix normalisation consists of the following phases: (a) Determining the minimum and maximum values in the matrix; (b) substracting the minimum from each element of the matrix, and dividing each element by the maximum; (c) computing the *elements norm* as the square root of the sum of each element power 2; and (d) dividing each matrix element by the *elements norm*. The program is tested with n = *30* iterations and input sizes from 1024×1024 to 4096×4096 to reach a memory-bound scenario.

5.4 Case study: Sobel

The Sobel Operator is described in 2. For this experimental study an implementation that iteratively processes frames from a video in YUV format has been chosen. The kernel was adapted from that of the Intel FPGA Support Resources, which received as input RGB images and transformed every pixel to a single channel on the fly. Instead, this transformation was removed, since the tested program reads an input video stream from a file, frame by frame and each frame has three components that are communicated to the device. Then, the Sobel filter is applied to each component launching the same kernel, once for each component. The resulting image is transferred back to the host to store it in an output video file. Each component of a frame is read, written, computed, or transferred separately. The input/output images are read from and written to files. The computation of the Sobel filter is a very fast operation. Thus, this study case is very demanding in terms of concurrency exploitation and asynchronous data-transfer executions. The program is tested with 120, 240, 300 and 360 frames of a high-definition video (Full HD images of 1920×1080 pixels) so the performance gained from computation and communication overlap could be perceived.

Algorithm 2 Sobel

```
1: function SobelOperation($A, B, threshold$)
2:     Data: x, y: column and row indices, respectively
3:     $h \leftarrow -A[y-1][x-1] - 2A[y-1][x] -$
           $A[y-1][x+1] + A[y+1][x-1] +$
           $2A[y+1][x] + A[y+1][x]$
4:     $v \leftarrow -A[y-1][x-1] + A[y-1][x+1] -$
           $2A[y][x-1] + 2A[y][x+1] -$
           $A[y+1][x-1] + A[y+1][x+1]$
5:     $temp \leftarrow \sqrt{(h^2 + v^2)}$
6:     if $temp > threshold$ then
7:         $B[x][y] \leftarrow 255$
8:     else
9:         $B[x][y] \leftarrow 0$
10:    end if
11: end function

12: function main
13:     Data: n: number of frames, A[3]: input video matrix, B[3]: output video matrix
14:     for $i \leftarrow 1, n$ do
15:         for $j \leftarrow 1, 3$ do
16:             MoveTo(A[j])
17:             SobelOperation(A[j], B[j])
18:             MoveFrom(B[j])
19:             LoadFrame(A[0], A[1], A[2])    ▷ Reads frame from source video and loads it in input matrices
20:             PutFrame(B[0], B[1], B[2])    ▷ Writes output matrices in destination video
21:         end for
22:     end for
23: end function
```

5.5 Experimental environment

In this experimental study the final prototype of the Controller library is compared against the reference programs in OpenCL with the synchronous and asynchronous implementations. The choice between synchronous and asynchronous policy can be done at run-time. The kernel codes are surrounded by the primitive *CTRL_KERNEL* and the accesses to global memory are rewritten with the Hitmap's *hit()* access macro.

The experimentation was conducted on nodes of the Intel DevCloud platform featuring FPGAs. Each of these nodes comprises an Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz with 192 GB of RAM equipped with an Intel Arria 10.

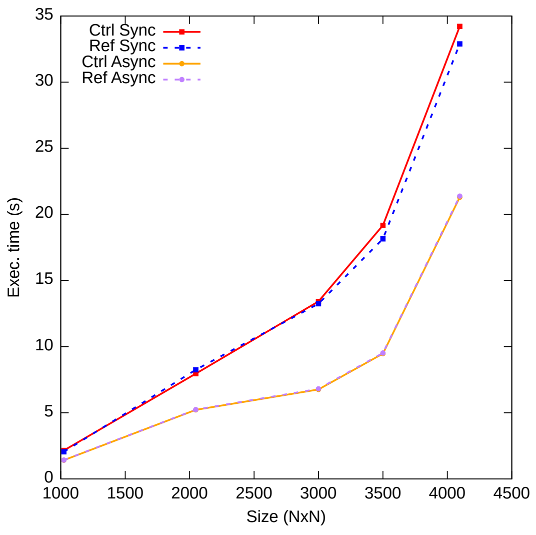
The O.S. is an Ubuntu 18.04 Linux distribution. All programs are compiled with GCC v7.4 using the Intel FPGA SDK for OpenCL. The performance experiments measure the clock time from the start of the first data transfer to the end of the last host-task. This includes data transfers, computation times, and system overheads.

5.6 Performance study

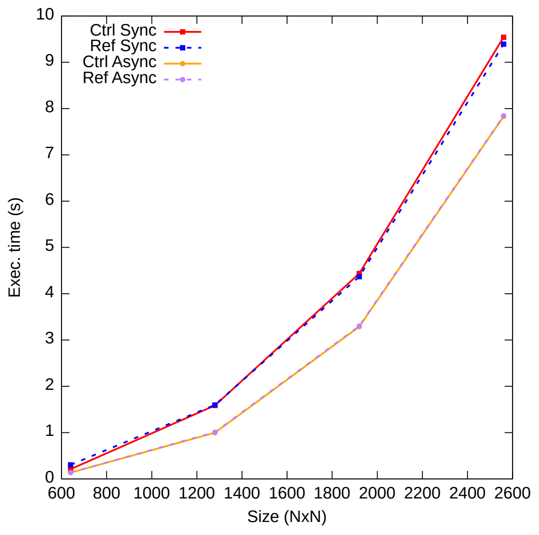
Figure 5.1 shows the performance results obtained for the different versions of the Hotspot and Matrix Power case studies. The plots represent the total execution times in the y-axis and different input data sizes in the x-axis. Figure 5.2 shows the performance of Hotspot, Matrix Power and Sobel filter when the number of iterations increases to enable a comparison with the latter, since input size is not a parameter. During the development of this work AOC was upgraded in Intel DevCloud, leading to some behavioural changes in the offline compiler and, as a result, some changes in the overhead of the Controllers library that are worth studying. For that reason, plots for the 19.3 and 20.1 version of the compiler are included. The same information is conveyed by tables 5.1 and 5.2 to ease the discussion of the results with numerical values.

From these results it is clear that the asynchronous policy of Controllers works specially well with FPGAs, with almost non existent overhead and, in some cases, even negative overhead. This can be explained through the treatment of events by Controllers, which processes them in batches, opposed to the reference programs, which do so once they are issued. The Controllers way reduces the number of requests to the device driver, hence leading to minor gains in time that could explain this counterintuitive results. The differences between the compiler versions are particularly noticeable in the Matrix Power and Sobel Filter kernels. It can be seen that for the 19.3 version the Matrix Power the Controller implementation achieves lower times than its counterpart, while the opposite happens when it is compiled with the 20.1 version of AOC. On the other hand, the opposite happens to the Sobel filter kernel, whose Controllers version performs better with the 20.1 compiler (notice that the difference lies in the asynchronous policy version, whilst for the synchronous one Controllers outperforms the native implementation regardless of the compiler).

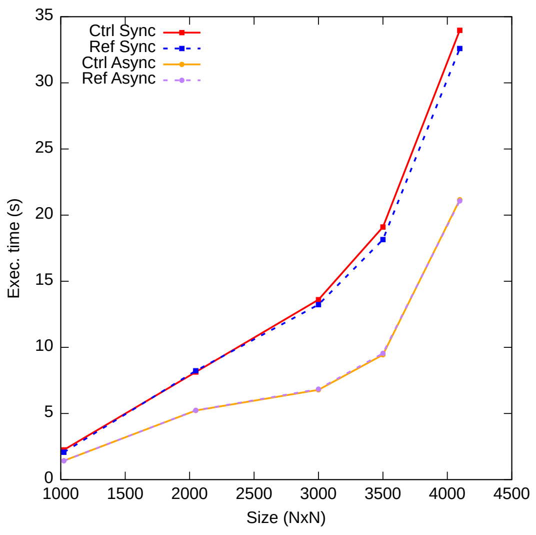
Recall that, although functionally the same, the code of Controllers kernels is not exactly the same once *CTRL_KERNEL* is expanded and hence the synthesised circuits differ. Table 5.3 gathers the operating frequencies of the synthesised kernels for both versions. They match the previous observations: the Matrix Power kernel achieves a frequency of 258.33 Hz in the Controllers implementation and 255.95 Hz in the reference one with the 19.3 version of AOC, whilst it was stated that the Controllers version ouperformed its counterpart with this compiler version. The opposite happens with the 20.1. The same reasoning can be applied to the Sobel filter kernel. However, the Controllers implementation always achieves lower times with the Hotspot benchmark regardless of the compiler, although according to 5.3 this sould only occur



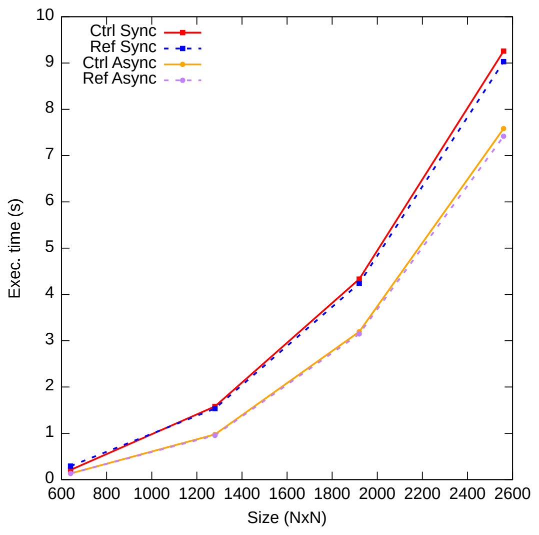
(a) Hotspot 400 iterations (v19.3)



(b) Matrix power 30 iterations (v19.3)



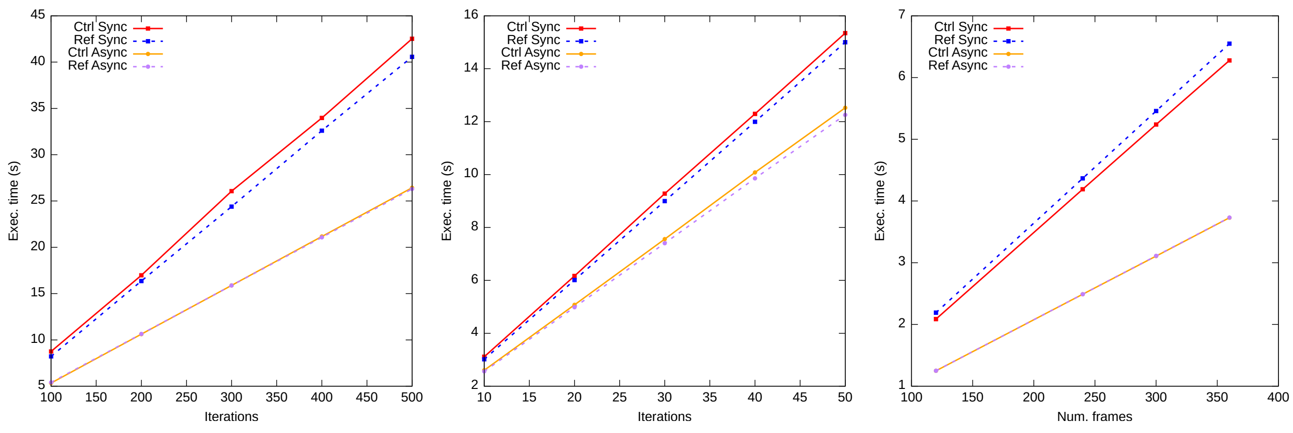
(c) Hotspot 400 iterations (v20.1)



(d) Matrix power 30 iterations (v20.1)

Figure 5.1: Experimentation results with Hotspot and Matrix Power benchmarks for different matrix sizes. AOC v19.3 results are at the top, whilst v20.1 are at the bottom.

with the 20.1 version and not in the 19.3 or, at least, it should not be that significant with this one. This suggests that other factors might be involved in this phenomenon. Unfortunately, profiling is of no help in this case, since the frequencies of the studied kernels change due to the instrumentation and the order relations between them changes (see, for example, how the Matrix Power kernel achieves higher frequency in the reference implementation with the 19.3 version of the compiler when it is compiled from profiling). Future work will include fixing the operating frequency of the kernel with the workaround described in [52], since the Intel FPGA SDK for OpenCL does not provide any native mechanism, to find out the underlying cause of this behaviour.



(a) Hotspot 4096×4096 matrix (b) Matrix Power 2560×2560 matrix (c) Sobel Filter 1920×1080p video

Figure 5.2: Experimentation results with Hotspot, Matrix Power and Sobel Filter for different number of iterations with AOC v20.3

AOC / kernel	v19.3	v20.1
Matrix Pow	258.33	271
Hotspot	361.11	315
Sobel	340.28	316

(a) Frequencies of Ctrl kernels

AOC / kernel	v19.3	v20.1
Matrix Pow	255.95	278
Hotspot	362.5	302
Sobel	345.83	314

(b) Frequencies of Ref kernels

AOC / kernel	v19.3	v20.1
Matrix Pow	250.0	271
Hotspot	310	292
Sobel	333.33	319

(c) Frequencies of Ctrl kernels (profiling)

AOC / kernel	v19.3	v20.1
Matrix Pow	257.58	242
Hotspot	360.80	307
Sobel	347.22	317

(d) Frequencies of Ref kernels (profiling)

Table 5.3: Frequencies (MHz) of the experimentation kernels in native (Ref) and Controllers (Ctrl) versions.

5.7 Development effort measures

This section analyses the differences in development effort between the Controller codes and the baseline implementation using OpenCL for the asynchronous scenarios. Four classical development effort metrics are measured: number of lines of code, number of tokens, McCabe's cyclomatic complexity [28] and Hasltead development effort [17]. The first two metrics measure the volume of code that the programmer should develop. The third metric measures the rational effort needed to program it in terms of code divergences and potential issues that sould be considered to develop, test and debug the program. For that, it considers the program as a connected graph and calculates the maximum number of linearly independent circuits. The last metric uses both code complexity and volume indicators to obtain a comprehensive measure of the development effort. The measured codes include the kernel definitions, kernel characterisation, the coordination host code, and data structures management. For a fair comparison, they have been formatted following the same criteria, with no line breaks in expressions or calls to functions, closing curly braces always on their own line, etc.

The results shown in table 5.1 indicate that programming using the new Controller library generates lower volume of code, a reduced cyclomatic complexity, and reduced Halstead measures than both, synchronous and asynchronous versions using OpenCL. This is specially noticeable for the asynchronous baseline versions, that introduce manually more complex mechanisms for kernel and data transfer synchronisations. These mechanisms are transparent and portable in the Controller programs. A close look at the codes indicates that the higer reduction is found in the parts of the host codes related to coordination, as expected.

5.8 Conclusions

The proposed solution integrates FPGAs in the Controllers library while incurring in little to no overhead. In fact, it has been found that in some situations, using Controllers can result in lower execution times than the vendor framework based on OpenCL. This is specially noticeable in the Sobel Filter case study. This phenomenon is likely to be due to a combination of a higher frequency of the Controllers synthesised kernel and the way our model manages events. Since Sobel Filter performs brief computations, differences due to events management is much more noticeable than with other benchmarks.

Programmability is highly increased, thanks to the reduction in the effort the programmer has to put to develop parallel applications in a heterogeneous system, as shown by the studied metrics, where our model scores significantly better results than the native OpenCL implementations.

Size	Impl.	Sync	Async	Overlap
1024	Ref	2.0487	1.4194	30.71 %
	Ctrl	2.1462	1.4120	34.21 %
Overhead		4.76 %	-0.53 %	
2048	Ref	8.2508	5.2437	36.45 %
	Ctrl	7.9550	5.2181	34.40 %
Overhead		-3.58 %	-0.49 %	
3000	Ref	13.2460	6.8106	48.58 %
	Ctrl	13.4202	6.7653	49.59 %
Overhead		1.32 %	-0.67 %	
3500	Ref	18.1536	9.5178	47.57 %
	Ctrl	19.1729	9.4718	50.60 %
Overhead		5.62 %	-0.48 %	
4096	Ref	32.8879	21.3831	34.98 %
	Ctrl	34.2106	21.2967	37.75 %
Overhead		4.02 %	-0.40 %	

(a) Hotspot

Size	Impl.	Sync	Async	Overlap
640	Ref	0.3037	0.1354	55.42 %
	Ctrl	0.2098	0.1363	35.04 %
Overhead		-30.90 %	0.70 %	
1280	Ref	1.5966	1.0084	36.84 %
	Ctrl	1.5868	0.9977	37.12 %
Overhead		-0.62 %	-1.07 %	
1920	Ref	4.3672	3.3014	24.40 %
	Ctrl	4.4355	3.2881	25.87 %
Overhead		1.57 %	-0.40 %	
2560	Ref	9.3919	7.8438	16.48 %
	Ctrl	9.5399	7.8294	17.93 %
Overhead		1.58 %	-0.18 %	

(b) Matrix Power

N fr	Impl.	Sync	Async	Overlap
120	Ref	2.1781	1.2502	42.60 %
	Ctrl	2.0640	1.2536	39.27 %
Overhead		-5.24 %	0.27 %	
240	Ref	4.3461	2.4901	42.71 %
	Ctrl	4.1539	2.4993	39.83 %
Overhead		-4.42 %	0.37 %	
300	Ref	5.4302	3.1103	42.72 %
	Ctrl	5.1928	3.1199	39.92 %
Overhead		-4.37 %	0.31 %	
360	Ref	6.5216	3.7305	42.80 %
	Ctrl	6.2145	3.7423	39.78 %
Overhead		-4.71 %	0.32 %	

(c) Sobel Operation

Table 5.1: Average execution times for Hotspot, Matrix Power and Sobel Operation compiled with AOC v19.3 (top left, top right and bottom, respectively). Percentage of overlap between synchronous and asynchronous versions and overhead of Controllers are also shown.

Size	Impl.	Sync	Async	Overlap
1024	Ref	2.0672	1.4217	31.23 %
	Ctrl	2.2510	1.4289	36.52 %
Overhead		8.89 %	0.51 %	
2048	Ref	8.2309	5.2391	36.35 %
	Ctrl	8.1363	5.2279	35.75 %
Overhead		-1.15 %	-0.21 %	
3000	Ref	13.2327	6.8397	48.31 %
	Ctrl	13.6032	6.7910	50.08 %
Overhead		2.80 %	-0.71 %	
3500	Ref	18.1535	9.5437	47.43 %
	Ctrl	19.0996	9.4395	50.58 %
Overhead		5.21 %	-1.09 %	
4096	Ref	32.5939	21.0722	35.35 %
	Ctrl	33.9702	21.1640	37.70 %
Overhead		4.22 %	0.44 %	

Size	Impl.	Sync	Async	Overlap
640	Ref	0.2915	0.1329	54.41 %
	Ctrl	0.2097	0.1341	36.04 %
Overhead		-28.05 %	0.94 %	
1280	Ref	1.5344	0.9557	37.72 %
	Ctrl	1.5809	0.9763	38.24 %
Overhead		3.03 %	2.16 %	
1920	Ref	4.2373	3.1461	25.75 %
	Ctrl	4.3323	3.1954	26.24 %
Overhead		2.24 %	1.57 %	
2560	Ref	9.0600	7.4305	17.74 %
	Ctrl	9.2370	7.6102	18.08 %
Overhead		1.95 %	2.42 %	

N fr	Impl.	Sync	Async	Overlap
120	Ref	2.1915	1.2507	42.93 %
	Ctrl	2.0871	1.2491	40.15 %
Overhead		-4.76 %	-0.12 %	
240	Ref	4.3668	2.4902	42.97 %
	Ctrl	4.1902	2.4897	40.58 %
Overhead		-4.04 %	-0.02 %	
300	Ref	5.4571	3.1112	42.99 %
	Ctrl	5.2389	3.1085	40.67 %
Overhead		-4.00 %	-0.09 %	
360	Ref	6.5488	3.7312	43.02 %
	Ctrl	6.2765	3.7290	40.59 %
Overhead		-4.16 %	-0.06 %	

Table 5.2: Average execution times for Hotspot, Matrix Power and Sobel Operation compiled with AOC v20.1 (top left, top right and bottom, respectively). Percentage of overlap between synchronous and asynchronous versions and overhead of Controllers are also shown.

Case study	Version	LOC	TOK	CCN	Halstead
Hotspot	Ctrl	224	1702	40	773603
	OpenCL Sync	336	2673	53	1497213
	OpenCL Async	401	3196	49	2036837
Matrix Power	Ctrl	145	1473	21	590244
	OpenCL Sync	239	2215	26	1205075
	OpenCL Async	307	2533	29	1470233
Sobel filter	Ctrl	137	1231	22	557772
	OpenCL Sync	202	1944	28	810726
	OpenCL Async	290	2561	38	1385085

Table 5.4: Measurements of development effort metrics for the reference and Controllers codes. It includes a comparison of number of lines of code (LOC), number of code tokens (TOK), McCabe's cyclomatic complexity (CCN) and the Hasltead's development effor metric (Halstead).

Chapter 6

Conclusions

This chapter introduces the following aspects:

- Definition of the objectives that have been fulfilled
- Future research lines
- Personal experience

6.1 Objectives fulfilled

This work is part of a research project directed by Trasgo research group. This work introduces an approach for the integration of FPGAs in the Controllers model, thus improving programmability and allowing automatic management of dependencies and seamless overlapping of data transfers and computation thanks to its asyncrhronous policy. In addition, its use leads to reductions in cyclomatic complexity, with significantly low overheads in the execution times compared to manually programmed and optimised solutions which directly leverage OpenCL.

1. Some of the existent works have been studied and compared to Controllers in section 2.2.4.
2. The Hitmap and Controllers libraries have been studied and a brief description of both is given in chapeter 2.
3. The design of the new backend is described in chapter 3 and its implementation in chapter 4.
4. The experimentation performed is explained in chapter 5 and enabled the validation of the model.
5. The experimentation allowed to reach conclusions about the improvement in programmability, overlap and the overhead caused by the use of Controllers with respect to the baseline native implementations.
6. The proposed model was validated through the experimentation conducted to that end, showing a proper integration of FPGAs in the model in a transparent way.

The result of this work has lead to a publication that has been accepted and will be presented in the international conference CMMSE 2020 (International Conference Computational and Mathematical Methods in Science and Engineering) on July 26th - 30th.

6.2 Future work

The development of this work has opened multiple research lines that might be the source of future works. In chapter 5 it was found that the use of Controllers leads to lower execution times than OpenCL in several scenarios. This could be partially explained by the difference in operating frequency between Controllers and reference kernels. This explanation had some holes, however, so more refined tools such as the workaround for fixing kernel frequency mentioned in [52] or the advanced profiling method detailed in [43] will probably be used to dig into the problem.

FPGAs allow more advanced techniques that are out the scope of this work and could potentially increase performance. This is the case of *partial reconfiguration*, which allows the modification of some parts of the synthesised circuit while the rest are runnning. Since programming time range from milliseconds to a second, there is a chance for overlapping them with computation or data transfers, leading to performance improvements. Combining this technique with the ideas for partial reconfigurable kernels in OpenCL in [42] could turn Controllers into an attractive option for shared heterogeneous environments in which context switches are frequent, such as cloud platforms.

HLS languages have the benefit of improving programmability and this has been the main focus of this work. However, the programmer might lose control over the synthesised circuit, relying on the design inferred by the compiler. This can be problematic for advanced use cases, as the one in 4.2.3. Fortunately, it is possible to integrate HDL modules with the Intel FPGA SDK for OpenCL, reaching full control over the final design.

6.3 Personal assessment

This work has been my first contact with research and the beginning of a career focused on research in HPC. Its development did not only allow me to learn new technologies but also introduced me to the research methodology. As a result, now I am capable of looking up the information I need to support my arguments more efficiently and I have become a better learner, which are valuable skills in every job.

I came from a data science background, so I had to learn many new concepts and tools to undertake this project successfully. It has been an exciting journey that has opened my mind and has given me the opportunity to work with FPGAs and expand my knowledge about electronics and the details of the low level layers of computers. Had it not been for this I would have never made a serious project with FPGAs and I would have not discovered all the learning oportunities it offers to electronics enthusiasts like me thanks to their reconfigurability.

All in all, this has been an enriching experience that reaffirms my motivation to pursuit a PhD and a career based on research.

Appendix A

Contents of the CD-ROM

The code of the backend elaborated in this work, as well as the case studies adapted for the experimentation, can be accessed upon request. For that, contact the members in charge at the Trasgo research group (`https://trasgo.infor.uva.es/`) for this work:

- **Arturo González Escribano**: arturo@infor.uva.es
- **Yuri Torres de la Sierra**: yuri.torres@infor.uva.es

Bibliography

[1] TOP500. https://www.top500.org/. [Online; accessed 8-June-2020].

[2] ISO/IEC 9899:201x; Programming languages – C. "http://port70.net/~nsz/c/c11/n1570.html#6.10.9", 2011. [Online; accessed 7-May-2020].

[3] A development sandbox for data center to edge workloads. https://software.intel.com/content/www/us/en/develop/tools/devcloud.html, 2020. [Online; accessed 15-June-2020].

[4] BertenDSP. GPU vs FPGA performance comparison. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays-FPGA'17*, 2016.

[5] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesus Labarta. Application Acceleration on FPGAs with OmpSs@ FPGA. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 70–77. IEEE, 2018.

[6] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[7] Cosmin Cernazanu-Glavan, Stefan Fedeac, Alexandru Amaricai-Boncalo, and Marius Marcu. Energy profiling of FPGA designs. In *2014 IEEE International Symposium on Robotic and Sensors Environments (ROSE) Proceedings*, pages 118–123, October 2014.

[8] IEEE Design Automation Standards Committee et al. Std 1076–2008, ieee standard vhdl language reference manual. *IEEE, New York, NY, USA*, 2008.

[9] Altera Corporation. Nios II Classic Processor Reference Guide. page 282.

[10] Altera Corporation. FPGA architecture (White Paper). 2006.

[11] Digi-Key. A-U200-P64G-PQ-G. https://www.digikey.es/product-detail/es/xilinx-inc/A-U200-P64G-PQ-G/122-2250-ND/9645681. [Online; accessed 13-June-2020].

[12] Digi-Key. A-U50DD-P00G-ES3-G. https://www.digikey.es/products/es?keywords=xilinx%20alveo%20u50. [Online; accessed 13-June-2020].

[13] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:7, 2007.

[14] Tony Gaddis. *Digital Fundamentals, Global Edition*. Pearson Education Limited, 2014.

[15] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1145–1154, 2013.

[16] Khronos OpenCL Working Group et al. The OpenCL Specification, version 1.0. 29, 8 December 2008.

[17] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[18] Mark Harris. How to optimize data transfers in CUDA C/C++. *NVIDIA Developer Zone*, 2012.

[19] Mark Harris. An efficient matrix transpose in CUDA C/C++. *Retrieved July*, 26, 2013.

[20] Hynix. 64Mb Synchronous DRAM based on 1M x 4Bank x16 I/O, 2007.

[21] Intel. Intel® Quartus® Prime Pro Edition Software. `https://www.intel.com/content/www/us/en/programmable/buy/design-software.html`. [Online; accessed 13-June-2020].

[22] Intel. *Intel FPGA SDK for OpenCL Programming Guide.* 2016.

[23] Intel. *Intel FPGA SDK for OpenCL Programming Guide.* 2019.

[24] Intel. *Intel FPGA SDK for OpenCL Best Practices Guide.* 2020.

[25] Qi Jia and Huiyang Zhou. Tuning stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 249–256. IEEE, 2016.

[26] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0.* Morgan Kaufmann, 2015.

[27] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[28] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[29] Michael McNamara. IEEE Standard Verilog Hardware Description Language. `https://www.verilog.com/IEEEVerilog.html`, 2008. [Online; accessed 9-June-2020].

[30] Ana Moreton-Fernandez, Hector Ortega-Arranz, and Arturo Gonzalez-Escribano. Controllers: An abstraction to ease the use of hardware accelerators. *The International Journal of High Performance Computing Applications*, 32(6):838–853, 2018.

[31] Raúl Nozal, Jose Luis Bosque, and Ramón Beivide. Enginecl: Usability and performance in heterogeneous computing. *Future Generation Computer Systems*, 107:522–537, 2020.

[32] David Padua. *Encyclopedia of parallel computing.* Springer Science & Business Media, 2011.

[33] Ioannis Parnassos, Nikolaos Bellas, Nikolaos Katsaros, Nikolaos Patsiatzis, Athanasios Gkaras, Konstantinos Kanellis, Christos D Antonopoulos, Michalis Spyrou, and Manolis Maroudas. A programming model and runtime system for approximation-aware heterogeneous computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[34] David A Patterson and John L Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface.* Morgan kaufmann, 2016.

[35] Artur Podobas, Hamid Reza Zohouri, Naoya Maruyama, and Satoshi Matsuoka. Evaluating high-level design strategies on FPGAs for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[36] Louis-Noël Pouchet and Scott Grauer-Gray. Polybench: The polyhedral benchmark suite (2011). *URL http://www-roc. inria. fr/˜ pouchet/software/polybench*, 2015.

[37] Arifur Rahman. *FPGA based design and applications*. Springer Publishing Company, Incorporated, 2008.

[38] Prasanna Sundararajan. High performance computing using FPGAs. Technical report, 2010.

[39] TerasIC. DE10-Pro. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=248&No=1144`. [Online; accessed 13-June-2020].

[40] TerasIC. DE5-Net FPGA Development Kit. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=526&PartNo=1`. [Online; accessed 13-June-2020].

[41] TerasIC. DE5a-Net-DDR4. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=228&No=1108&PartNo=1`. [Online; accessed 13-June-2020].

[42] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117, Dublin, Ireland, August 2018. IEEE.

[43] Anshuman Verma, Huiyang Zhou, Skip Booth, Robbie King, James Coole, Andy Keep, John Marshall, and Wu-chun Feng. Developing dynamic profiling and debugging support in OpenCL for FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[44] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-based computing systems with OpenCL*. Springer, 2018.

[45] Chao Wang, Xi Li, Junneng Zhang, Peng Chen, Xiaojing Feng, and Xuehai Zhou. FPM: A Flexible Programming Model for MPSoC on FPGA. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 477–484, May 2012.

[46] Wayne Wolf. *FPGA-based system design*. Pearson education, 2004.

[47] Xilinx. Vivado Design Suite - HLx Editions. `https://www.xilinx.com/products/design-tools/vivado.html#buy`. [Online; accessed 13-June-2020].

[48] Xilinx. *SDAccel Programmers Guide*. 2019.

[49] Victoria Zhislina. Why has CPU frequency ceased to grow? *Intel. Retrieved October*, 14, 2015.

[50] Hamid Reza Zohouri. High performance computing with FPGAs and OpenCL. *arXiv preprint arXiv:1810.09773*, 2018.

[51] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420. IEEE, 2016.

[52] Hamid Reza Zohouri and Satoshi Matsuoka. The Memory Controller Wall: Benchmarking the Intel FPGA SDK for OpenCL Memory Interface. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 11–18. IEEE, 2019.

[53] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 153–162, 2018.