Adapting a HPC runtime system to FPGAs

Georgios Christodoulis

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Georgios CHRISTODOULIS

Thèse dirigée par **Frédéric DESPREZ**
et codirigée par **François Broquedis**
et **Olivier MULLER**, MCF, Grenoble INP

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Adaption d'un système HPC pour intégrer des FPGAs

Adapting a HPC runtime system to FPGAs

Thèse soutenue publiquement le **5 décembre 2019**,
devant le jury composé de :

Monsieur FREDERIC DESPREZ
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Directeur de thèse
Monsieur CHRISTIAN PEREZ
DIRECTEUR DE RECHERCHE, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Rapporteur
Monsieur SMAÏL NIAR
PROFESSEUR, UNIV. POLYTECHNIQUE DES HAUTS-DE-FRANCE, Rapporteur
Monsieur FRANÇOIS BROQUEDIS
MAITRE DE CONFERENCES, GRENOBLE INP, Examinateur
Monsieur OLIVIER MULLER
MAITRE DE CONFERENCES, GRENOBLE INP, Examinateur
Monsieur RAYMOND NAMYST
PROFESSEUR, UNIVERSITE DE BORDEAUX, Président
Monsieur DAVID NOVO
CHARGE DE RECHERCHE, CNRS DELEGATION OCCITANIE EST, Examinateur

Abstract

Along with the traditional CPU cores, processing units of different architectures are now employed by the High Performance Computing (HPC) community in order to obtain improved efficiency and performance. A Field Programmable Gate Array (FPGA), is a hardware fabric composed of interconnected re-programmable logic and memory blocks. This type of processing unit, is considered a promising candidate to amplify the efficiency and the computational power of heterogeneous HPC platforms since it comes with massive parallelism and a reduced amount of abstraction layers between the application and the actual hardware.

However, exploiting FPGA requires an in-depth knowledge of low-level hardware design and high expertise on vendor-provided tools, which is not aligned with the skills of high performance computing application programmers. In the scope of this thesis, we have designed a framework that allows straightforward development of scientific applications over heterogeneous platforms enhanced with FPGA. Using this framework requires only a limited knowledge of the underlying architecture, and an FPGA can be used in the same way as any other type of processing unit. At the core of the proposed framework there is the StarPU heterogeneous runtime system, that was extended to support this new type of accelerator, hiding from the programmer complex operations deriving from the complexity of the underlying architecture while it allows fine control of the performance through different scheduling strategies. For the communication part, we created Conor, a communication library based on RIFFA, that enforces consistency in scenarios with concurrent accesses to the FPGA.

The approach proposed by our framework is evaluated across two directions. The first one corresponds to programmability, while the second one concerns the performance overhead imposed by the additional components attached to the FPGA. Both the programmability and overhead of the framework are evaluated using a basic blocked version of matrix

multiplication showing the ease of development and the negligible overhead imposed by FPGA management to the rest of the framework. In addition to our experiments on matrix multiplication, we created an efficient hardware design of *GEMM*, that will allow the execution of more complex and interesting applications like the Cholesky decomposition.

Contents

List of Figures

Chapter 1

Introduction

The evolution of technology has lead to computing systems of tremendous performance capabilities coupled with a fair amount of complexity. As an example, the IBM Summit super-computer reaches a theoretical peak performance of $200\,\mathrm{Pflop\,s^{-1}}$, at the expense of a great increase in both architectural complexity and energy consumption. In order to overcome the complexity associated with such machines, researchers have developed models and tools that provide clean interfaces to developers bridging the gap between a common modern computer and such a system.

To overcome the energy demands required for this amount of computations, the community has oriented itself towards heterogeneous processing resources. The first example of truly high performing heterogeneous computations was the one of the GPU, that showed orders of magnitude of performance increase in certain kinds of computation compared to the traditional CPU. The idea behind heterogeneity is to identify and decompose an application into different parts with similar demands and bottlenecks and assign them to the type of processing unit that corresponds best to their characteristics. FPGA are re-configurable pieces of fabric, that lately gained the interest of the high performance computing community for their low energy consumption and potential for augmented performance. From a high level point of view, those properties derive by the fact that in an FPGA, the description of the application and its actual hardware implementation are closer (reduced amount of abstraction layers) than a traditional architecture like the CPU.

The trade-off of utilizing FPGA for general purpose processing is the complexity of hardware design. Despite the fact that hardware vendors have invested a lot of effort to enable the configuration of those devices using a higher-level language, there are still plenty of challenges to overcome in order to obtain efficient implementations. Those challenges are mainly due to the massively parallel nature of the hardware, as well as the fact that development happens in the inverse way compared to a traditional

CPU. The fundamental processing units of an FPGA are very primitive, yet in the order of thousands and hundreds of thousands, so the level of parallelism exceeds by far the one that software developers are used to. On top of that, in the process of hardware design, it is the processing unit that is being adapted to the algorithm, and not the other way around.

All things considered, only a fraction of the high performance computing community considers programming FPGA for now. Besides the requirements for fine low-level control of hardware design, programmers need to handle mechanisms associated to the management of the device. The main objective of this thesis is to improve the accessibility of heterogeneous architectures containing FPGA accelerators to parallel application programmers. To do so, we propose a framework which enables developers to use FPGA as another accelerator within a traditionally heterogeneous high performance computing platform. We focused on ease of development and portability without sacrificing performance, to position this work as a first step towards bridging the gap between the hardware and high performance computing communities. Our solution relies on an extension of the StarPU heterogeneous runtime system and a new communication library called Conor that facilitates the communication between the software side and the device through PCIe. On the hardware side, we provided the control units responsible for the integration of the hardware tasks that will be provided by the developer This complexity has been the challenging, motivating and rewarding part of this thesis along with the fact that the product of this work is directly applicable to real large scale systems.

This manuscript is organized as follows:

- Chapter 2 provides the essential information about the background required for the understanding of this work. Section 2.1 introduces the concept of parallelism and its twofold nature, the spatial and the temporal one. Section 2.2 explains the different levels of complexity of a modern high performance computing system, and how parallelism is exploited in every one of them, while section 2.3 introduces FPGA as an accelerator to such system.

- Chapter 3 provides the different approaches associated to this work by the software (3.1) and hardware (3.2) community respectively.

- Chapter 4 introduces our framework as well as the first series of its evaluation. More precisely, section 4.1 focuses on both the software and hardware level

integration we went through during this thesis. In this chapter, we also evaluate the behavior of our framework on homogeneous execution scenarios, using a blocked version of matrix multiplication (4.2). The evaluation is across two dimensions, one being the programming effort that our framework requires for the support of FPGA (section 4.2.2), as well as the performance overhead of our integration (section 4.2.3).

- Chapter 5 presents our work towards truly heterogeneous experiments. On the first part of the chapter (in section 5.1) we present a series of experiments, where we used the same blocked version of matrix multiplication used in chapter 4, that multiplication tasks are executed simultaneously on the CPU and the FPGA. For a more complete example of heterogeneous execution, we introduce Cholesky decomposition, an application that is assembled by four different kernels, one of which (*GEMM*), we have chosen to optimize and synthesize in hardware (section 5.4).

- Lastly, chapter 6 presents a summary of this study as well as its possible perspectives.

Chapter 2

Background And Problem Statement

Contents

The core of this PhD stands across two well-established research areas: high performance computing and hardware design using FPGAs. This chapter introduces the reader to the notions and concepts of both, that are necessary for the further comprehension of the contribution and the results of the work. Section 2.1 is an introduction to the concept of parallelism highlighting both its spatial and temporal nature. Section 2.2 introduces high performance computing systems presenting their hardware organization and the way they are programmed today. Section 2.3 is dedicated to FPGAs and provides details on both their architecture and how to exploit them from the programmers' point of view. Section 2.4 then presents a taxonomy of high performance computing applications and a suitability analysis of FPGAs to each class, based on the literature. The chapter ends with section 2.5 that provides

an analysis of the challenges and our positioning regarding the integration of FPGAs in HPC.

2.1 Parallelism Is Everywhere

Once the capabilities of a single core reached a ceiling because of physical limits of the matter, mainly its ability to drain off heat, parallelism was employed in order to further augment the performance of computing platforms. The purpose of high performance computing is to deliver fast computational solutions to problems and algorithms of multiple fields. Often, those problems are by nature parallel, meaning there are parts of the computational load that can be executed independently and hence simultaneously with others. It is of vital importance to identify these parts in order to exploit the underlying processing resources to reach the best possible performance. Fundamentally parallelism has a twofold nature, the temporal and the spatial one. In the following analysis, we present two generic techniques to capitalize on its both aspects: pipelining for the temporal parallelism and resource duplication for spatial parallelism. Those two universal concepts can be applied to almost every problem. To demonstrate them we use the example of clothes-washing. Along with the concepts of pipelining and resource duplication, we introduce fundamental performance evaluation metrics like the throughput and the latency.

Problem description: the laundry Assume a given set of dirty t-shirts that needs to be delivered clean and well folded. The standard procedure involves:

- Step a: doing the actual washing (processing unit: washing machine)
- Step b: drying the clothing (processing unit: drier)
- Step c: ironing and folding (processing unit: ironing table)

The latency metric corresponds here to how much time it is required in order to get one t-shirt clean. Throughput refers to how many t-shirts per time unit we can deliver (assuming we have more than one t-shirt to process). In a completely sequential paradigm, we would wait for a t-shirt to pass from all the three stages (washing, drying, ironing and folding) before getting the next one into the procedure.

Pipelining The first optimization technique that comes immediately in mind is that we can overlap washing with drying and ironing among different t-shirts. This is the idea of pipelining that exploits temporal parallelism. The concurrent utilization of the three different processing units on different t-shirts is possible because the three procedures are independent and can operate individually. In other words, the effort that is needed in order to perform the entire execution can be split into a set of individual and well defined steps.

We can at this point evaluate the impact of pipelining using the latency and throughput metrics introduced above. This will lead to a series of observations and remarks regarding the parameters that effect the behavior and the effectiveness of the method. Let step k requires t_k units of time for its completion. First by evaluating the sequential version of the laundry we obtain a latency and a throughput of:

$$Sequential\ latency = t_a + t_b + t_c$$

$$Sequential\ throughput = \frac{1}{t_a + t_b + t_c}$$

By evaluating the pipelined version we obtain:

$$Pipelined\ latency = t_a + t_b + t_c$$

$$Pipelined\ throughput = \frac{1}{max(t_a, t_b, t_c)}$$

The first observation here is that pipelining does not effect the latency. In other words pipelining is a technique that aims to increase the throughput. The second observation is that the effect on the throughput is correlated with the balance among the time spent on each one of the steps. On the one hand, if the steps are well balanced in terms of time demands, the gain in throughput is proportional to the number of steps. On the other hand, if the majority of the processing is spent during one of the steps, then the throughput of the pipelined version converges to the throughput of the sequential one. Said differently, the effectiveness of pipelining is determined by the delay of the slowest step compared to the average step delay.

Resource duplication The second intuitive optimization technique is to get equipped with multiple units of each stage (laundry machines, driers and ironing tables) so that we can perform the same step multiple times concurrently, exploiting the spatial nature of parallelism. This optimization technique is called resource duplication and by default is also a throughput optimization strategy. For our laundry problem, as well as for any problem where every workload will be processed by every step, resource

duplication can increase the throughput proportionally to the number of duplicates if we have an equal amount of duplicates for every stage. In a laundry with three washing machines, three dryers and one iron table, the throughput is the same than the one of a laundry with one single processing unit per stage. This is because all the t-shirts will have to serialize on the ironing step.

Assuming a facility with x processing units per step, the latency and the throughput are the following ones:

$$Duplicates\ latency(x) = t_a + t_b + t_c$$

$$Duplicates\ throughput(x) = \frac{x}{t_a + t_b + t_c}$$

Resource duplication is also technique to increase the throughput, by adding supplementary processing units. The bottleneck of the throughput increase lies at the step of the procedure with the least amount of resources.

Combining the two techniques As one could expect, the above two techniques can be combined together in order to exploit both temporal spatial parallelism. In our laundry problem, the facility can be equipped with a series of machines for every stage, and the operation on the machinery can occur in a pipelined fashion. This would increase the overall throughput in two ways, the amount of t-shirts processed per time unit will be proportional to the number of available resources per unit, and the time per new t-shirt will be decreased to the maximum delay per processing step. The final throughput for this combined approach where pipelining is combined with x duplicates is the following one:

$$Throughput(x) = \frac{x}{max(t_a, t_b, t_c)}$$

2.2 HPC Systems

The size and the computational power delivered by modern supercomputers are now tremendous. The highest ranked computer system of today, in the TOP500 list called *Summit*, according to the last published data of November 2018, holds 2,397,284 IBM POWER9 22C cores at 3.07 GHz and 2.8 PB of main memory [119]. The system is also equipped with NVIDIA Volta GV100 GPUs and connected with Dual-rail Mellanox EDR Infiniband. Its theoretical peak performance is 200 $\mathrm{Pflop\,s^{-1}}$, and its submitted power consumption is 9,8 MW. This tremendous power of computation comes with very complex hardware and memory layouts that HPC application developers need to

fully comprehend in order to exploit such platforms at their full potential. This section first presents the hardware architecture of HPC systems. Moreover, it demonstrates how an algorithm can be decomposed and transformed in order to adapt to the underlying parallel infrastructure, from the point of view of the HPC programmer. Finally, this section classifies from a very high level, existing programming models currently used in order to implement a parallel algorithm.

2.2.1 Hardware Organization

This section presents the typical hardware organization of an HPC system such as the *Summit* one mentioned above. It decomposes the system to different levels, and highlights the hardware parallelism on every one of them. Our analysis will start at the CPU level in section 2.2.1.1. In section 2.2.1.2 we present how several CPUs are combined together assembling a node, while in section 2.2.1.3 we present how a several nodes are combined together to form a cluster.

2.2.1.1 Parallelism Inside A CPU

A CPU is a processing unit responsible for the execution of streams of instructions of a fixed architecture that are called programs. An instruction is a set of bits, that combined with an architecture called the Instruction Set Architecture (ISA), performs a specific operation that can be of various types such as integer or floating point arithmetic operation, branching or memory access. In most of the cases, this set of bits can be decomposed in two fields, the operation code (opcode) and the operands. From the programmers point of view, a stream of instructions is executed sequentially. That is, instructions are executed one after the other.

In practice, a CPU employs both pipelining and resource duplication to increase throughput when executing a stream of instructions. For the better understanding of how these two techniques are employed in a modern CPU, we will use the architecture of figure 2.1 as a reference. The architecture is called GoldmontPlus and is used in modern low-power x86-64 CPUs by Intel.

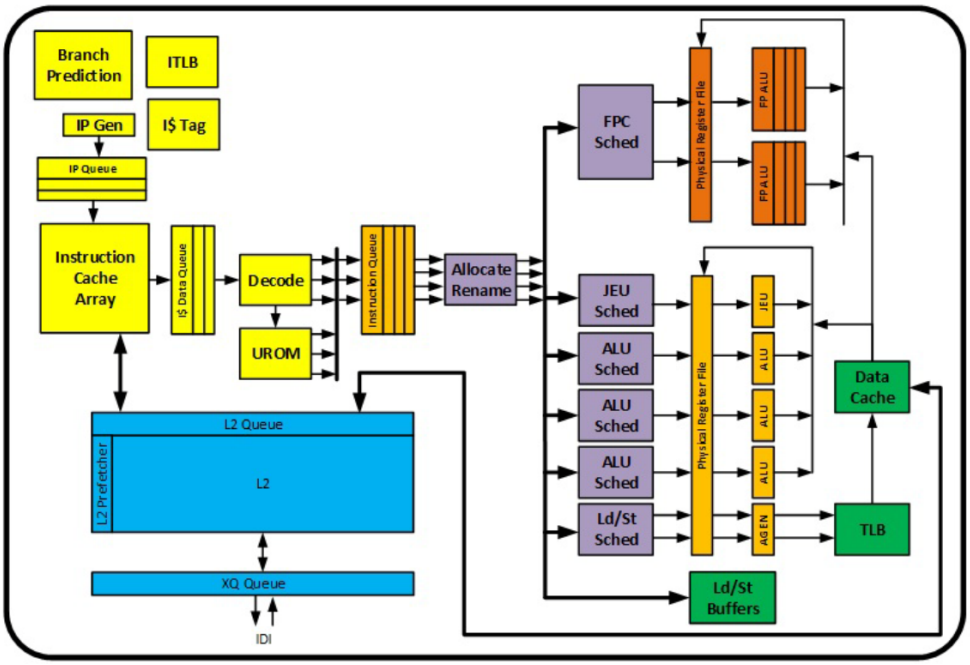


Figure 2.1: Overview of the GoldmontPlus CPU architecture by Intel [2]. The architecture is used as a reference to demonstrate how parallelism is employed in the design of a modern CPU.

Program's instructions live in memory, and are fetched as the execution progresses according to a program counter, a CPU register pointing to the next instruction to execute. Between memory and the processor there are several layers of caches.What we see in figure 2.1 (yellow part) though is that an instruction is fetched from a dedicated cache (Instruction Cache Array) and then passed to a queue in order to be further executed.

The next step is the decoding of the instruction. It corresponds to the procedure of the instruction's identification by its opcode and fetching of its operands. In the GoldmontPlus architecture, the decoding phase corresponds to a control operation, during which the instruction passes from the Instruction Cache Array to the Instruction Queue (orange part). At that time, the control unit also specifies which execution path will later be followed during the execution phase.

The next step that we see in figure 2.1 is related to the register renaming (purple part), which is a technique related to the out-of-order execution of instructions, and goes beyond the scope of this analysis. Nevertheless,[110] provides a complete presentation of register renaming techniques.

The next phase in the lifetime of an instruction is the execution. According to its type, an instruction associated to an arithmetic operation will be forwarded to the

appropriate processing unit that can be either a floating point unit (FP ALU) or an integer arithmetic and logic unit (ALU).

Once the result of the corresponding operation is available, it will be written in the physical register file (the same series of registers that hold the input of the instruction).

The paths exploited so far correspond mainly to instructions with computational nature. There other types of instructions, like those responsible for the interaction with memory (loading and storing) or control oriented ones (branches). Although, what is important to keep for the rest of the analysis is the fact that the life of an instruction can be decomposed to the phases of fetching, decoding, execution, interaction with memory and write back.

Instruction Pipelining In our laundry example of section 2.1 we saw that pipelining is a technique that allows to exploit temporal parallelism and obtain increased throughput. It is the same very same basic idea, that has been employed by CPU designers, in order to deliver better performance, and in that scope is called *instruction pipelining*. In the presentation of the GoldmontPlus architecture, we observed that simply put, the lifetime of an instruction can be decomposed in five phases: fetching, decoding, execution, memory accessing and writing back. Instruction parallelism allows the overlapping of execution of different stages of different instructions at a given time. This can happen since every stage is independent, and there are different units responsible for its execution.

Overlapping the execution of instructions have some side effects that require some additional control to ensure the correctness of the computations. Consider a scenario according to which two consecutive instructions are pipelined, where the output of the first is used as an input on the second one. Such phenomenon if often referred to as *data dependency conflict*, and is usually handled by stalling a part of the pipeline so that by the input reading phase of the second instruction the output of the first one is the valid one. In such cases, the performance of the processor can converge down to the performance of a non-pipelined one, but this only depends on the dependencies of the instructions' stream.

Super-Scalar Pipelines The second technique of our laundry example of section 2.1 was resource duplication, that gives the ability of simultaneous execution of multiple workloads of the same kind. In the example architecture of figure 2.1 we see multiple execution units for basic arithmetic (ALUs) operations. A pipeline including

such duplicates is called *superscalar*, since multiple instructions can be in the same pipeline stage concurrently, the same way multiple t-shirts can be washed at the same time in a laundry with several washing machines.

Exploiting parallelism in the processor with the aforementioned techniques allows an execution throughput greater that one instruction per cycle, which is the case for nearly every modern CPU. Because of the level that this parallelism is exploited, it is often referred to as *Instruction Level Parallelism* (ILP).

In processor with super-scalar pipelines, specific attention should be paid to minimize the amount of time that processing units remain idle. Techniques like out-of-order issue, branch prediction, speculative execution, register renaming and out of order execution have been developed, but exceed the scope of this study, whose main purpose is to highlight the role of parallelism in a modern high performance computing system at every level.

Multi-core CPU The previous paragraphs presented how parallelism is expressed inside a CPU made of single processing unit, referred to as a *core*. To further exploit spatial parallelism, several cores are combined together in modern CPU. The CPU cores have access to the same shared main memory. In such a multi-core CPU, several instruction streams can be executed concurrently, and they can communicate through the shared memory.

2.2.1.2 From Processors To Nodes

The previous section linked the fundamental concepts of parallelism introduced in section 2.1 to a typical modern CPU architecture. This section focuses on the introduction of the next level of complexity of modern parallel computers, which is the node level.

To further exploit spatial parallelism, several processing units are combined together to make a *node*. Along with the CPUs, other kind of processing units have been employed in order to enhance the processing capabilities of the node. Figure 2.2 shows a typical node design. The so called *host* part of the node is made of one or more multi-core CPU (4 in the example) accessing the same main memory. The other part of the node contains a series of accelerators often connected to the host via a PCIe bus. In this design, accelerators do not share the same memory as CPU cores, hence they have their own privates memories.

The number of CPU cores can greatly vary in modern processing nodes, depending on computing needs. Typical sizes are four (e.g. a single 4-core CPU) to thirty-two

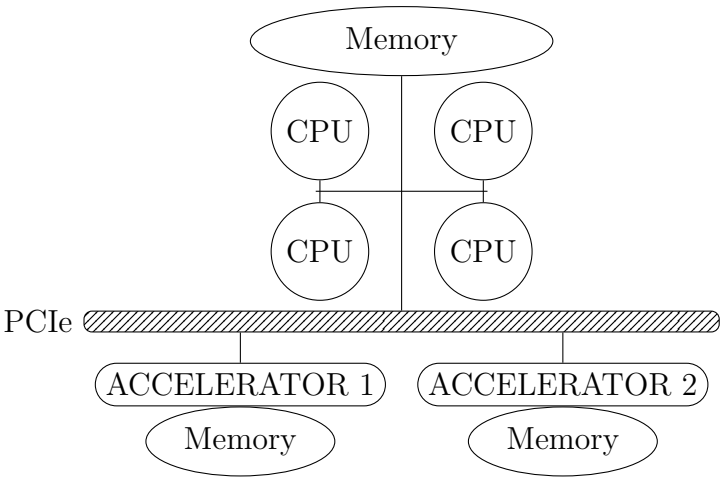


Figure 2.2: Reference structure of a typical heterogeneous processing node. The node is constructed by a set of CPU accessing the same main memory (host), and a series of accelerators connected to the host via a PCIe bus. Typically, accelerators do not share the same memory as CPU cores, hence they have their own privates memories.

cores (e.g. four 8-core CPU). In a node with a single CPU all the cores access the same single shared memory as already stated. In other words they have uniform memory access at the same memory banks.

The increasing amount of CPU cores, is shown to throttle the performance of memory operations for the uniform memory accessing scenarios. To solve this issue hardware designers introduced architectures where all the CPU cores share a physically distributed memory. In this case the access to memory is no more uniform. Such architectures are called Non Uniform Memory Access (NUMA) architectures. There is a dedicated circuitry responsible for the exchange of data between the CPU cores and the different parts of the memory. The size of such architectures can increase significantly, with usual cases of nodes having a total of 256 cores (e.g. sixteen 16-core CPU) or more. Fine control over the location of data when performing memory accesses is then crucial to obtain the best performance that can be delivered by such architecture. The deeper understanding of their structure and behavior though, lies beyond the focus of this work.

On the accelerators side, a typical node is coupled with one or two GPUs. Many-core processors, such as the Intel Xeon Phi, are also nowadays used as accelerator. The parts of the computation that are more efficiently executed on an accelerator than on a CPU are offloaded from the host side to the accelerator one. These accelerators come with there own private memories as already stated. The consistency between the host memory and the private memories of the accelerators is not handled

by the hardware. Considering as an example the architecture presented in figure 2.2, and assuming that initially data lie in main memory, we can study the hypothetical scenario according to which a part of the execution will occur in one of the GPUs. The corresponding data need to be transferred from the main memory to the private memory of the accelerator. Meanwhile, an authority needs to ensure which of the two copies is valid at a given moment of the execution, as well as to synchronize the non-valid copies when needed.

2.2.1.3 From Nodes To Clusters

Like multiple processing units can be put together to form a parallel, possibly heterogeneous compute node, the same approach can be followed to design large-scale clusters of nodes, an extreme -in terms of capabilities- instance of which is Summit introduced in section 2.2. The building block of a cluster is the node and nodes are connected using a network, like ETHERNET or INFINIBAND .

The interconnection of multiple nodes brings enormous processing power that comes to the theoretically unlimited scalability of such a system, although there are challenges to overcome deriving from the interconnection itself, along with the management of the underlying resources.

From the software point of view, there was a need to establish a protocol for the communication of execution units that are being processed on different nodes. This communication's overhead usually varies according to the distance of the processing units in the cluster. Another major concern in order to obtain the desired performance is the load balancing between the different nodes; in other words, we need to ensure that the workload will be distributed to the available processing nodes in a way that the variance of the completion time for every node is minimal as well as the communication overhead required for the exchange of data between nodes. Along with load balancing, another development concern is to set the proper affinity of execution units to nodes or cores (usually ensuring that an execution unit will remain close to the data that is associated with, making a better use of data locality). In other words, the developer needs to create and maintain a mapping between the processing load to a node, or more even more specifically to a core, in order to obtain a minimal overhead deriving from data movement.

2.2.2 How To Program An HPC System?

In the preceding part of the chapter, one could obtain a global view of the hardware architecture of a modern high performance computing platform. It is of vital

importance for the programmer to capitalize on the amount of computational power provided by such systems, that mainly derives from the underlying parallelism. To do so, one needs to adopt strategies, concepts and methodologies that would allow the concurrent execution of a formerly sequential algorithm or of a new one. Programming models and associated infrastructures are then needed for this parallel algorithm to be expressed and executed.

2.2.2.1 Extracting Parallelism In An Application

Given a reference architecture, the entry point for the development of high performing application is the analysis and understanding of the algorithm that will later be expressed in the form of a program. In order to take advantage of a massively parallel platform like the ones described in section 2.2.1, it is essential that the nature of the computations is also heavily parallel. If not, the entire workload will be processed by a single processing unit while the rest of the resources will remain idle.

Finding Concurrency In the case the application contains parallelism, it is not always trivial to identify it in order to design a parallel version of the algorithm. Task and data decomposition, defined as follows, can be used to decompose the problem into pieces that can execute concurrently:

- The task decomposition dimension views the problem as a stream of instructions that can be broken down into sequences called *tasks* that execute simultaneously. For the computation to be efficient, the operations that make up the task should be largely independent of the operations taking place inside other tasks.
- The data decomposition dimension focuses on the data required by the tasks and how they can be decomposed into distinct chunks. Strong computational dependencies among the different chunks make the overall computation inefficient.

Task and data decomposition are tightly coupled, since one implies the other, although by making them distinct, we make the design emphasis explicit. In some cases the problem will naturally break down into a collection of tasks, hence the task-based approach is easier. In most cases though, the tasks are difficult to isolate and the decomposition of the data is a better starting point. Regardless of whether the starting point is a task or a data-based approach, a parallel algorithm ultimately needs individual work loads that will execute concurrently, so the tasks must be

identified. In the case only data-parallelism is used, the tasks will all perform the same computation, but on different chunks of data.

2.2.2.2 Parallel Programming Models

Many parallel programming models have been proposed and are in used to implement a parallel algorithm on an High Performance Computinghardware architecture. The evolution and proposal of these models have been following the evolution of hardware architectures. Combinations of these models are often used in order to exploit the hardware parallelism at all the levels.

Shared memory programming models In a shared memory programming model, the programmer specifies parallel activities communicating through writes and reads in a shared memory. Depending on the particular programming model, the proper synchronization between the activities is handle by the programmer or automatically. Among many others, POSIX threads and OpenMP are two widely used shared memory programming models. They are described in the next two paragraphs.

POSIX threads In the POSIX threads programming model, several independently controlled execution paths, called threads are created by the programmer. POSIX threads are implemented as a library with functions for creating, destroying and coordinating thread activities. This model is especially appropriate for the fork/join parallel programming pattern, according to which several execution flows are instantiated and then merged when everyone is completed. Threads are sharing the dynamically allocated heap memory causing programming difficulties regarding synchronization . When multiple threads access shared data, they must be properly synchronized by the programmer to avoid race conditions and deadlocks. That corresponds to a task oriented parallel programming model for shared memory architectures. The number of threads is not necessarily related to the number of processors.

OpenMP OpenMP is a shared memory programming model providing a higher level abstraction than Posix threads. Said differently, it eases the creation and management of threads from the programmers point of view. It is a portable application programming interface (API) implemented as a combination of a set of compiler directives (thread creation, synchronization, memory management), pragmas, and runtime providing both management of the thread pool and a set of library routines. Parallel regions enables a set of instructions to be replicated across a set of threads.

OpenMP is specially suited for the loop parallel program structure pattern, although the Single Program Multiple Data (SPMD) and fork/join patterns also benefit from this programming environment.

Message passing programming models In message passing programming models, several entities are evolving in separate memory address spaces and communicate via explicit message exchanges. This is a natural model for a distributed memory system, where communication cannot be achieved directly by sharing variables.

Message Passing Interface (MPI) is the most used standard in this category. It defines a portable message passing interface designed to function on a wide variety of parallel computing architectures. It is a library that specifies the names, calling sequences and results of the subroutines or functions to be called. In MPI, work load partitioning and task mapping are assigned to the developer. In addition to point to point exchanges, MPI supports collective communications.

Programming models for heterogeneous architectures The previously described programming models focused on programming homogeneous architectures. The introduction of GPUs by Nvidia and their exploitation to non-graphic uses, that is General Purpose GPU (GPGPU), led to programming models explicitly dedicated to heterogeneous architectures including GPUs. CUDA from Nvidia, and the OpenCL standard are the two most used programming models in this category but many other solutions such as OpenMP target construct [25], OpenACC [76] or HMPP [33] have been proposed to ease the programming of architectures with GPUs.

In contrast with shared-memory, with these programming models, the programmer needs to offload some data to and from accelerators private memories. Also, because parts of the application are offloaded to accelerators, the programmer needs to write the code to be executed there. In CUDA for example, this done by writing C code which is compiled to GPU binary with the Nvcc compiler provided by Nvidia.

Because programming heterogeneous architectures is the main focus of this thesis, we extensively describe programming models for heterogeneous architecture in the related work chapter 3.

2.3 FPGA

In the previous section we explained how and why processing elements of different architectures were employed for general purpose computing in order to enhance the

performance of high performance computing platforms. Our work has focused on incorporating FPGAs in such platforms. From that point of view, FPGAs can be considered as reprogrammable integrated circuits. In other words, they are hardware fabrics, with interconnected basic elements of logic and memory, whose content can change dynamically according to the configuration.

In order to explore the behavior and characteristics of FPGAs we will go along the natural process one will follow to integrate such an accelerator to a given heterogeneous platform. There is a series of fundamental questions to be answered:

- How to program the device?
- How is memory organized?
- How does the device communicate with the rest of the platform?

In order to understand how an FPGA is programmed, we need to understand its architecture which is presented in section 2.3.1. Then, section 2.3.2 presents the process of low level hardware design, as well as the utility of higher level synthesis tools that can be used in order to program the accelerator. In section 2.3.3 we present the different types of memory a modern FPGA chip and board are equipped with, and how they are used by modern vendor tools. Lastly, section 2.3.4 describes how an FPGA is connected with its environment to assemble an heterogeneous execution platform.

2.3.1 FPGA: Architecture

When asked to integrate an FPGA, understanding its architecture is crucial, since it is not similar to the one of a traditional CPU that can be programmed using an ISA like x86. For the configuration of FPGAs, programming happens at a much lower level, that in practice it is no longer considered software development but hardware design.

Figure 2.3 shows a high level overview of an FPGA. It is composed by Configurable Logic Blocks (CLB), memory blocks (BRAM) and an interconnection network (PI). In this section we will dive into the CLB components of the chip, while memory will be analyzed in section 2.3.3.

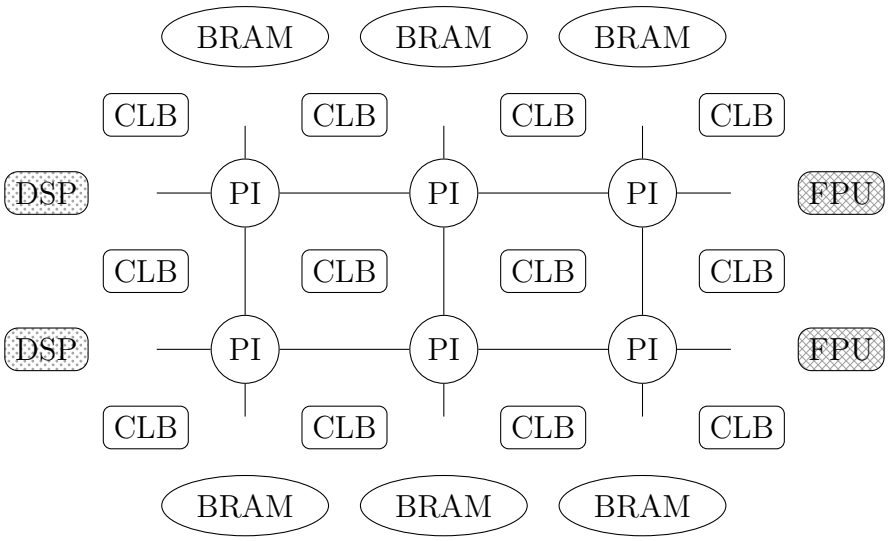


Figure 2.3: Overview of the internal architecture of FPGAs. Configurable logic blocks (CLB) are connected with embedded memory (BRAM) and dedicated processing elements (DSPs and FPUs), in an array-like arrangement.

As a reference of a CLB we will use the Zynq 7000 FPGA family from Xilinx [1], since this is the FPGA family we conducted our experiments on. Figure 2.4 shows that a CLB is composed out of two disconnected slices. Each slice has an input and an output port (CIN and COUT respectively), and is also connected to the interconnection network (marked as Switch Matrix). A simple slice contains *function generators*, *storage elements*, *carry logic* and *multiplexers*, and is called SLICEL. Along with SLICEL slices, there are also SLICEM, that can additionally provide data storing functionalities through distributed RAM elements. A CLB can be either assembled by two SLICEL slices, or by one SLICEL and one SLICEM.

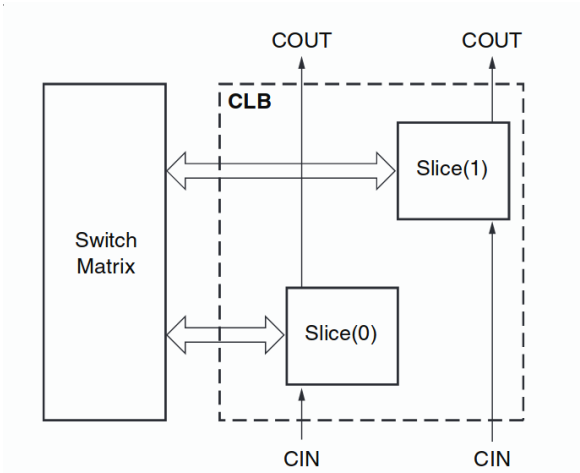


Figure 2.4: Overview of the internal structure of a CLB for the Zynq 7000 family of FPGAs by Xilinx [7]. Every block is assembled by two slices, that can put through either purely logic operations, or logic operations along with storage. The logic operations are provided via logical function generators implemented by LUT and can simulate the behavior of any Boolean function.

Function generators are the building blocks of logic, and are implemented using *look-up tables* (LUT). Every slice contains four function generators, every such generator is implemented by a LUT and can behave as an arbitrary Boolean function.

Along with the logic, a CLB is associated with some control signals like set/reset, write-enable, clock and clock-enable. Set/reset initialize the internal state of the LUT, so that we can obtain the output of the Boolean functions given the current inputs. Write enable is used in order to enable writing on the memory elements of the CLB. The clock is a signal that is used to synchronize the lower level hardware components like the flip-flops used to compose the LUT or the memory elements.

Along with the traditional reconfigurable logic, provided by the LUT, FPGA vendors have tried to augment the capabilities of the chips on dedicated computations. In order to deliver high performance on basic arithmetic operations Digital Signal Processors (DSPs) are added, while some modern FPGAs come also with dedicated logic for floating point (FPU) operations.

It is useful at this point to provide an order of magnitude of the amount of logic elements of a chip, and for that we will use our experimentation platform (7VX690T) as a reference. There are ∼108 k CLB slices out of which ∼65 k slices are of type SLICEL and the rest (∼43 k) are of type SLICEM. To implement those slices ∼400 k LUT have been utilized. This particular chip is not equipped with a dedicated unit for floating point operations, although it holds 3.6 k DSP slices organized in 18 columns of 200 slices per column.

2.3.2 How To Program An FPGA Chip?

In section 2.3.1 we introduced a reference architecture of an FPGA, that we described as an integrated circuit that can be reconfigured. A configuration of the chip, is a hardware design that determines the contents of the CLB, the memory as well as the way that those components are connected via the interconnection network. The default languages for hardware design are called hardware description languages, Verilog and VHDL being the two representative examples of the category. Programming in such language is happening in very low level, justifying the existence of two distinctive communities: software developers and hardware designers. In appendix A and listing 20 one can see a template code of a single bit-adder module written in Verilog. The purpose of this template code is to demonstrate how low-level hardware design is compared to software development even for a language like C, that is particularly known for the fine control it gives to the programmer.

Given a hardware description of a design, there is a series of steps to be followed so that we have an FPGA circuit behaving accordingly. First the hardware design needs to be synthesized, which involves the procedures of syntactic checking and optimizations to the given FPGA chip. Next step is placement and routing, which is the procedure of mapping the synthesized design to the logic blocks and memory elements of the given chip, as well as determining the behavior of the interconnection network. Once placement and routing is finished, we can obtain the hardware design in a bitstream file, that will be flashed in the device and will give the requested behavior. All those functionalities, are provided by vendor tools that are proprietary, allowing very limited flexibility to the designers. This flexibility mainly relies on the ability of the designer to determine the exhaustiveness of the heuristics used by the algorithms (that by default are NP-complete).

In order to ease the accessibility of FPGA, so that they become usable for general purpose computing, researchers from both communities have created synthesis tools that bridge the gap between higher level languages (C, C++, python) and the hardware description level. Such tools are called High Level Synthesis (HLS) tools, and have become a matter of study for many years in an academic and industrial level. Nowadays, HLS tools can deliver augmented performance, providing fine control of the design in the form of compiler directives, that allow the programmer to exploit the massive parallelism of the FPGA, increasing their popularity and usability.

2.3.3 FPGA: Memory

Memory is a fundamental component of every processing unit, including FPGAs. From a high level view there are three types of storage associated to an FPGA chip, the memory units inside a CLB, the block memory (BRAM) and the on board memory (external to the chip). Every type of the above comes at a different level of integration, and has different capacity and accessibility overhead.

Block level memory, has already been introduced in the presentation of the CLB in section 2.3.1. Data here are mainly stored in the LUT and in one bit registers. LUT memory is a versatile kind of storage, although when selected to store big memory structures the final design will be huge (in spatial terms). This will make the rest of the synthesis process (place and design and routing) very hard in terms of complexity and hence time consuming. Nevertheless, it is the selected memory type to store simple variables by most of the synthesis tools.

The block memory (BRAM) is a series of blocks of RAM, coupled with CLB, in most FPGAs in order to host large data structures. This is the type of memory that will concern developers the most, because most of the currently used High Level Synthesis tools map by default complex structures like arrays to this type of storage. BRAM memory is organized into single or dual port blocks, resulting in a reading throughput of one or two words per cycle respectively. Developers using HLS tools are equipped with a series of optimization directives that allow the fine control of how data are mapped to BRAM. Efficient block memory control combined with the massive amount of parallel resources available can deliver impressive performance increase. The capacity of the available BRAM memory will be a factor that indicates the granularity of the kernels, and should be taken into account from the beginning of the design process.

On top of the memory structures mentioned above, most of the modern FPGA boards, allow a memory extension with a DDR3 or DDR4 RAM sets. The characteristics of those memory blocks are similar to those used for conventional systems like laptops and barebones. In order to communicate with this external board memory, vendors provide a series of control designs, that should be manually tuned and incorporated to the rest of the design. This task used to be complicated for people without a certain amount of expertise on hardware design. Nowadays vendors have worked a lot to make the design process much simpler on more recent versions of the synthesis tools.

To provide an insight on the difference in capacity between the different storage elements of an FPGA we will use again as a reference the chip we conducted our

experiments on (7VX690T). Inside the CLBs there are ∼10 Mb of distributed memory and ∼5 Mb of shift registers. With the BRAM blocks, the chip reaches a storage capacity of ∼53 Mb. Regarding the on board memory, we can slot two blocks of 4 GB external DDR3 memory, enabling the hosting of much bigger structures.

2.3.4 FPGA: Communication With Its Environment

In order to use FPGAs in the context of general purpose processing, they need to be connected with their environment, so that they can receive computations' input and propagate their output. Modern FPGAs are usually delivered into boards with a rich amount of IO ports. The fabric itself can be directly coupled with a processor (most of the times a single or dual core ARM processor) via an AXI bus, where the FPGA chip and the CPU cores co-exist on the same chip, that is they make up a System on Chip (SoC). As an alternative to that, the FPGA chip can be connected to a host machine via one of the available buses, typically the PCIe bus.

The block diagram of figure 2.5 shows the peripherals accompanying the FPGA chip, which the board we conducted our experiments on (VC709- 7VX690T) is equipped with.

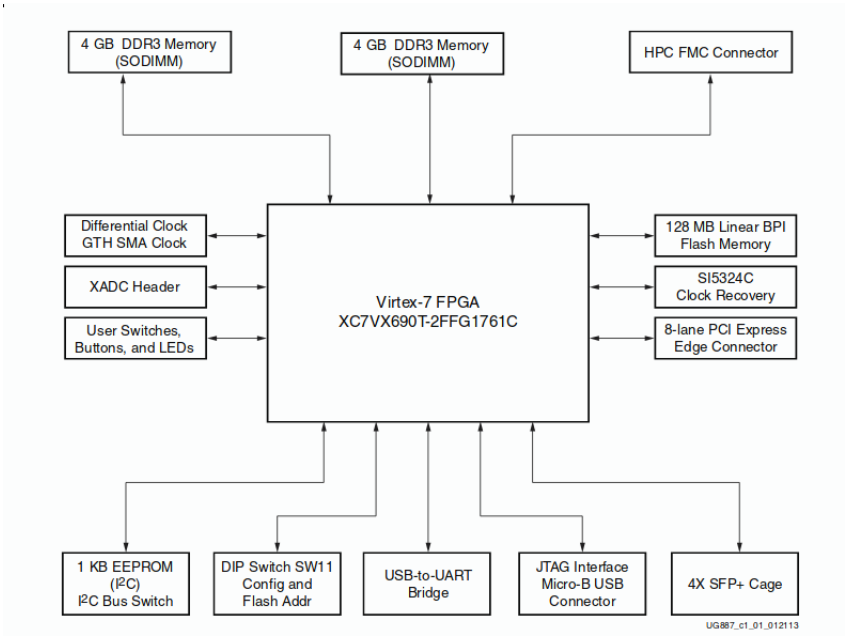


Figure 2.5: Block design of our experiment board - VC709, a member of the Zynq 7000 family from Xilinx [6]. The design is used to present the environment of the chip on the board. Among with the DDR3 blocks, the Differential Clock, the DIP Switches the UART and the JTAG interfaces, we highlight the 8-lane PCIe Edge Connector that allows the communication of the chip with the host node.

A PCIe based communication facilitates fundamental principles of high performance computing like the scalability and modularity. Building a system based on the SoC approach mentioned earlier, restricts the developers to a small amount of CPU cores and a single FPGA chip. On the other hand, utilizing the PCIe as the connection mean, allows a greater level of complexity on the node level. A node is not limited to a single FPGA device.

2.4 Performance Results Of FPGAs On HPC Applications

Despite the effort from the hardware community to bridge the gap between software development in HPC and hardware design required in order to exploit reprogrammable hardware, one could infer from the previous sections that there is still a fair amount of complexity associated to the usage of FPGAs in that context. The motivation to overcome this complexity barrier is associated to the ability of FPGAs to deliver noticeable performance with very low energy requirements. The energy consumption of an FPGA is significantly lower than the one of a typical CPU a fact that derives from the overhead imposed by the levels of abstraction between the CPU and the software languages utilized. Performance wise, according to the literature FPGAs can deliver comparable if not greater performance compared to a typical CPU, for certain applications.

HPC applications are scattered among a wide spectrum of application domains from commercial to financial, medical or scientific fields. Nevertheless, Asanovic et al. [16] has sorted them in different classes, putting together applications with similar behavior and bottlenecks, in a taxonomy called Berkeley categorization. The initiative of the following analysis is to present reported artifacts of the behavior of FPGA in every such category, following the analysis of Escobar et al. [38]. Roughly, in the aforementioned study, HPC applications are classified in the following categories:

- Dense Linear Algebra
  One of the most popular category, that includes decomposition and factorization methods (Cholesky, LU, QR). BLAS, CUBLAS, LAPACK are libraries widely adopted by developers, offering outstanding performance using block and tile algorithms to perform the aforementioned calculations. State of the art FPGA implementations of applications from this group, have shown impressive performance results, although CPU-GPU combination is still more than an order of a magnitude faster [116, 128, 130].

- Sparse Linear Algebra
  In this category we get the same group of applications as before, though in this cases matrices have few non-zero elements. We observe irregular memory accesses, low data reuse and low number of floating operations per access. Efforts to implement Sparse Matrix Vector (SpMV) multiplication in FPGAs have shown promising results, comparable with the performance of CPU-GPU platforms [34, 81, 88, 101].

- Spectrum Methods
  This class of applications contains arithmetic computations on the frequency domain, partial or ordinary differential equation solvers being the two most representative candidates. Most of the algorithms employ a Fast Fourier Transform. The most promising results for this class are coming from the 3D-FFT version, where studies show that an FPGA implementation can outrun a GPU-CPU combination [61, 121].

- N-Body Methods
  It is the set of algorithms describing the interactions of the particles due to gravitational forces between each other. The CPU profiles appear irregular memory accesses and scalability issues. CPU-GPU solutions increased a lot the performance over CPU only solutions, although they consume up to fifteen times more power per floating operation [58, 70, 109] than FPGA solutions.

- Structured Grids
  It is the class of problems where data can be arranged into arrays with interacting neighboring elements. They are characterized by regular memory accesses. Studies have shown acceleration potentials using FPGAs for this class of applications, although most of the times it is a theoretical estimation of the expected performance [72, 107].

- Unstructured Grids
  Here applications are characterized by irregular memory accesses due to their natural inability to be formatted in a structured representation. Techniques like data compression and encoding have been used to tolerate the bandwidth bottleneck between the host and the FPGA, but the results are comparable to a middle class CPU [10].

- Map Reduce
  It is a set of distributed algorithms without strong dependencies allowing maximum parallelism. The most significant paradigm of this set is Hadoop which assumes a full dedicated cluster control, which makes it not the best candidate for HPC. There are several attempts to combine Map Reduce algorithms but each one of them requires a careful analysis to evaluate the results [120, 131].

- Combinational Logic
  It is a class of application with extensive work in FPGAs invested in. It includes cryptographic algorithms like AES, DES, RC4, SHA-1, RSA. In multiple studies optimization techniques have been suggested, with results close to the ones obtained by CPUs [39, 86, 112].

- Graph Traversal
  It is a category of applications from several domains, since a graph is a flexible data structure to represent pieces of information and dependencies between them. BFS and DFS are the two widely known approaches to traverse a graph, and BFS is selected as a Graph500 benchmark, a clear notation of its importance. It suffers from poor locality caused by excessive data access to fairly irregular memory footprints. Studies have shown impressive FPGA performance compared to powerful CPUs [17, 23].

- Dynamic Programming
  It is a class of applications showing a high level of dependencies, scattered among a wide range of scientific fields. FPGA development relies mostly in the field of bioinformatics [108], using systolic arrays and pipelining. In a series of studies we see acceleration obtained by FPGAs over CPUs; although the serious bottleneck of those works is that the recursive nature of DP requires complex management units with low portability [90].

- Backtrack and Branch&Bound
  It is a computational class with very few results in the area of the FPGAs, with some interesting results though over the Pentium CPU family. It is composed by algorithms that can be split into a branching and a pruning face in order to obtain the result of the computation. Distinctive behavioral attribute of the class is the irregularity and unpredictability of the data footprint There are very few FPGA implementations for this class.

- Graphical Methods
  This class contains applications like neural or Bayesian networks, where a graph is employed in order to represent and explain the behavior of a system. In this class we have a show case of an FPGA implementation (on Virtex-5) of a Bayesian learner that vastly outperforms the combination of CPUs and GPUs. Neural network implementations on FPGAs have also shown interesting results with serious acceleration over commodity CPUs [71, 79, 95].

- Finite State Machines (FSM)
  The executional footprint of this class, is an oscillation through different states, where the computational load is to determine the next state as well the outputs of the current one. Latency and memory size is the two basic bottlenecks for this category with a few attempts to deploy FSMs in FPGAs published.

The preceding classification indicates that HPC applications show different executional patterns that are more or less suitable for different processing units. FPGAs can be a promising alternative both in terms of absolute performance and energy efficiency. The goal of a truly heterogeneous platform is to harness this diversity, by leveraging from the advantages of each computational resource.

2.5 Problem Statement

The landscape of common HPC architectures is currently moving to large-scale heterogeneous platforms, similar to those introduced in section 2.2. These machines are composed by a large number of interconnected nodes, similar to those of figure 2.2 of section 2.2.1.2. Traditionally, those nodes contain multiple CPU cores on the host side, and a couple of GPUs on the accelerator side. However, to build exascale-ready supercomputers, the HPC community now also considers other kinds of processing units that can perform better, both in terms of time and energy consumption. Based on the reported artifacts about the performance of FPGAs in traditional HPC applications of section 2.4, we believe they can be one of them.

Unlike other kinds of accelerators where the application is optimized to fit to the architecture of the processing unit, FPGAs work the other way around; it is the underlying hardware that is tailored to fit the applications' needs. Despite the promising perfromance and energy efficiency, there is a big amount of complexity that comes together with this type of processing units. The complexity is not only limited to the programming effort required for the development of hardware designs

of high performance, but also to the effort required to orchestrate the execution of an application in such heterogeneous environment.

Figure 2.6 shows the reference node architecture we are targetting in this work. The FPGA is connected to a node via a PCIe bus, the same way another accelerator like a GPU does. We believe that an integration attempt will only be successful and widely accepted if FPGAs are handled the same way as the rest of the processing units of the node. This way of integrating FPGAs would allow to benefit from a wide range of mature software stacks from the HPC community and to allow true heterogeneous execution.

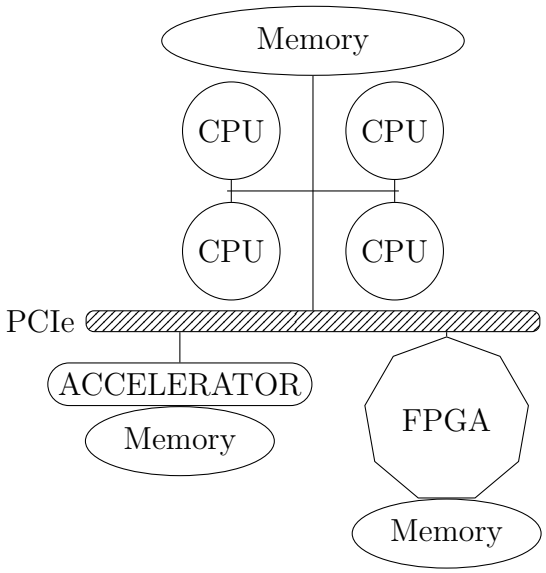


Figure 2.6: Paradigm of our reference heterogeneous platform. A typical node of our focus is assembled by a series of CPU cores accessing the same physical memory (host), coupled with an accelerator (typically a GPU) and an FPGA. The devices assembling the node are connected over PCIe with the host. They have their own physical address space, hence they are represented with their own private memories in the design.

In respect to the previous analysis, the objective of this work is to deliver:

- A runtime system that provides an adaptable (easy to plug and drive to an existing platform), scalable (multiple FGPA devices per node), portable (not limited to a certain board), non-restrictive (FPGA support does not prevent the concurrent use of other accelerators) and transparent (the developer does not need to be aware of the underlying architecture details) task-level integration of FPGAs in a Distributed Memory architecture, based on a relaxed consistency memory model that provides a global address space to the programmer.

- A case study were an FPGA is used to accelerate parts of an application, showing the impact of the integration to the rest of the parameters taken into account from the scheduling point of view.
- An example of a widely used HPC kernel - *GEMM*, that is optimized to deliver augmented perfromance when executed on an FPGA.

Chapter 3

Related Work

Contents

There has been a significant amount of contributions from both the HPC and the hardware communities that constitute the background of this work. To accompany the description that follows, Figure 3.1 shows how we classify existing approaches.

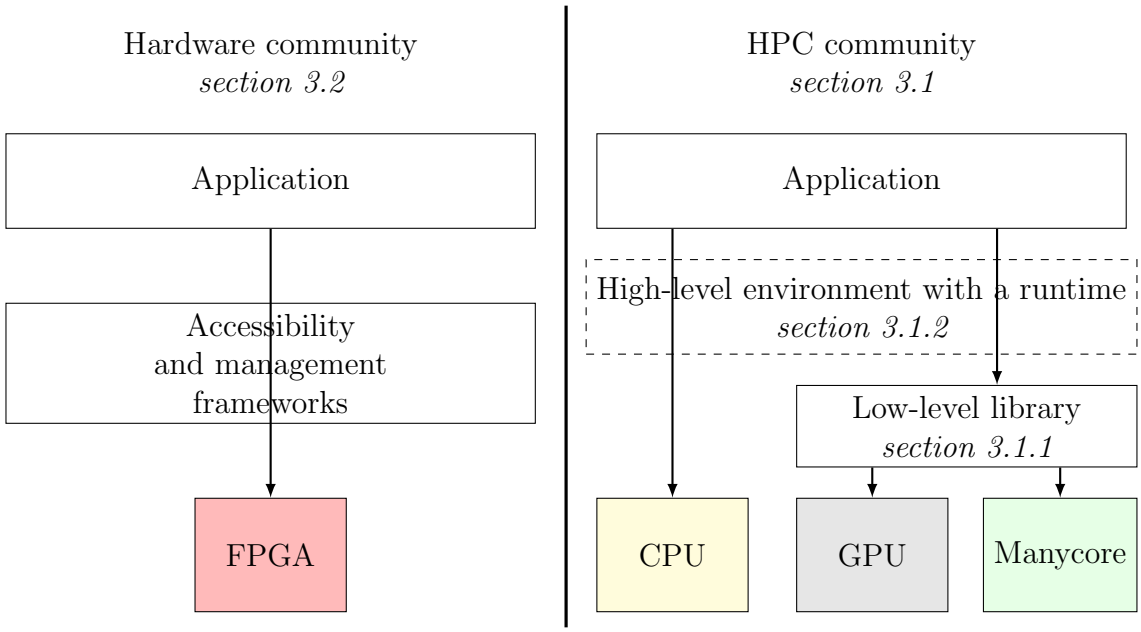


Figure 3.1: An overview of the related work on programming environments for heterogeneous platforms. On the high performance computing community side (section: 3.1), we first introduce low-level libraries that allow the execution of code on accelerators (section 3.1.1) and then higher-level runtime environments (section 3.1.2). On the hardware community side, we introduce frameworks that ease the accessibility and management of FPGAs (section 3.2). Our taxonomy distinguishes three classes of works: the thread based ones (section 3.2.1), the task based approaches (section 3.2.2) and the process based ones (section 3.2.3).

Researchers from the HPC community have worked extensively in order to provide programming models and frameworks that allow the programmers to exploit the available parallelism in heterogeneous machines mainly consisting of CPUs and GPUs. Section 3.1 presents a summary of these programming models and frameworks.

Meanwhile, researchers of the hardware community have worked towards improving accessibility of FPGA from the software side. While most of those attempts succeeded to provide a framework that allows a software programmer to offload code to an FPGA, those approaches did not manage to get widely accepted. Approaches from this community were either focused on restrictive programming models or provided a limited level of abstraction. Section 3.2 gives an overview of representative examples of this class of frameworks. Our taxonomy distinguishes three classes of

works: the thread based ones (section 3.2.1), the task based approaches (section 3.2.2) and the process based ones (section 3.2.3).

Section 3.3 presents the few works, as our presented in this thesis, trying to combine approaches proposed by both communities.

Finally, section 4.3 discusses and positions our work regarding existing approaches.

As mentioned previously, in order to assemble this section, representatives of two large communities need to be presented. In order to maintain a reasonably readable chapter, we tried to restrict ourselves to approaches that have been active or have significantly contributed to the evolution of the domain. Moreover, the information presented is based on the best of our understanding of the available literature, and not on exhaustive experimentation, since such a procedure would require a lot more time than a PhD thesis.

3.1 Heterogeneous Programming Approaches From The HPC Community

Once GPUs started to be adopted for general purpose processing, standards have been proposed in order to ease the application development. The two most widely accepted programming standards are CUDA and OpenCL. CUDA was introduced by Nvidia for their graphics processors while OpenCL is the non-proprietary open standard that provides equivalent functionalities to the programmer.

OpenCL and CUDA provide the means for the programmer to access the device. They come with an API for the communication between the host and the device memory, as well as mechanisms to orchestrate the execution of the device code. Nevertheless, there is a big amount of low level concerns remaining to the programmer, that are not automatically handled by the standards. The programmer needs to ensure the correctness and efficiency of its applications, managing mechanisms like the workload scheduling and memory management. In section 3.1.1 we present these two standards the way they were introduced for the GPU case, while in section 3.3 we introduce approaches that extend them to support FPGA.

Dedicated programming environments were developed to handle automatically, or at least more easily, and efficiently the aforementioned procedures. These environments come along with a so-called *runtime system*. This term happens to be overloaded across the researchers depending on their perspective and their scope. Within the scope of this work, a runtime system is a framework that is responsible for the

management of the dependencies between the parallel code regions, the data movement between the different memories of the platform and the scheduling/mapping of the workload to the resources. The vast majority of such heterogeneous environments rely on CUDA or OpenCL for data exchange between the GPU and the host side, as well as for the orchestration of the part of the computation happening on the devices, on top of which they provide support for high-level functionalities. Section 3.1.2 introduces these environments.

3.1.1 Low-Level Device Programming Libraries: OpenCL and CUDA

As already mentioned above, OpenCL and CUDA are two standards, a proprietary one provided by Nvidia and the open equivalent, that provide a generic mechanism to access accelerators, originally developed for easing the utilization of graphics processors for general purpose computing. They both provide programming interfaces and hardware abstractions enabling developers to accelerate applications with data-parallel computations in an heterogeneous computing environment consisting of CPUs and GPUs. An OpenCL or a CUDA program is usually made of a *host* part executed by one or more of the CPUs, and some computing *kernels* designed to be executed on the GPUs. While the host part follows the standard path of compilation, the GPU computing kernels have to be compiled using a specific compiler.

Since the two models are very similar, OpenCL being the open equivalent of CUDA, we will stick to the first and its terminology during our analysis. OpenCL as a standard specifies a platform model, an execution model, and a memory model.

According to OpenCL, a platform consists of the *host* (CPU) and the *device* side, the last being further decomposed into one or more compute units, each one assembled by one or more processing elements.

Every OpenCL device, is associated with a command queue, that can be used from the software side to enqueue kernels, to transfer data and orchestrate the synchronization of kernels execution.

The execution of a kernel is associated to a context, which is the combination of a device, the kernel function, a program executing on the host and memory objects. The synchronization of the kernels happens either via specific commands (like barriers) or events. During its lifetime an OpenCL kernel passes through different states. First it is *queued*, which is being enqueued to a command queue, then it is *submitted*, that is being flushed from the command queue to a device and waits in this state until it becomes *ready*, meaning that all of its dependencies are met. A ready kernel goes to

running state during its execution, by the completion of which it is considered *ended*. Lastly, it goes to a *completed* state, by propagating a *CL_COMPLETE* event.

A single context may contain several command-queues, and the programmer is responsible for the orchestration of the execution, by creating events for the synchronization of the kernels among the different queues.

The organization of the working units in OpenCL is mostly influenced by the architecture of GPUs. At the lowest level, we have the working units. Each one of these units is represented in OpenCLby an autonomous threads of execution with a global identifier called a work-item. Working items are further organized into work-groups. Synchronization primitives can only be used among working units of the same work-group.

In OpenCL there are two types of memory, the *host* one, which is memory accessible from host software, and the *device* one which is the one accessible from the kernel function executing in the accelerator. Device memory, is organized into different layers, starting with the *private* memory, which is storage facility attached to and only accessible by a work-item. On the immediately higher level, there is *local* memory, that is associated to a work-group. On top of that there is *global* memory, that is accessible by every work-item of every work-group of the context. A part of global memory is characterized as *constant*, and is guaranteed to remain constant during the execution of a kernel instance. Global memory may or may not provide caching structures, depending on the architecture implementing the API.

OpenCL implements a shared virtual memory mechanism, so that kernels executing in the device and host parts of the program can share the same address space. This mechanism is the global device memory and the host memory. The synchronization between the two is orchestrated explicitly by the programmer using map/unmap commands.

The code snippets of listings 1 to 3 show a C-based pseudo code of the host side program that performs matrix multiplication. The code sources are based on the open implementation provided by the University of Eindhoven [5].

```
// OpenCL device memory for matrices
cl_mem d_A, d_B, d_C;

// Allocate and initialize host memory for matrix A
unsigned int size_A = WA * HA;
unsigned int mem_size_A = sizeof(float) * size_A;
float* h_A = (float*) malloc(mem_size_A);
randomInit(h_A, size_A);
// Same for B and C

// Initialize Environment
cl_uint dev_cnt = 0;
clGetPlatformIDs(0, 0, &dev_cnt);

cl_platform_id platform_ids[100];
clGetPlatformIDs(dev_cnt, platform_ids, NULL);

// Connect to a compute device
clGetDeviceIDs(platform_ids[0], gpu ? CL_DEVICE_TYPE_GPU :
↪  CL_DEVICE_TYPE_CPU, 1,
&device_id, NULL);

// Create a compute context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// Create a command commands
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Listing 1: C-based pseudo-code of matrix multiplication in OpenCL, environment initialization.

In the very beginning of the execution, matrices of the device side are declared while matrices on the host side are declared, allocated and initialized. Then the user needs to detect the number and the IDs of connected devices, create their context and their command queues.

```
// Create and build the kernel program from the source file
LoadOpenCLKernel("matrixmul_kernel.cl", &KernelSource, false);

program = clCreateProgramWithSource(context, 1, (const char **) &
↪  KernelSource, NULL, &err);

clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
kernel = clCreateKernel(program, "matrixMul", &err);
```

Listing 2: C-based pseudo-code of matrix multiplication in OpenCL, device code compilation.

According to the standard, OpenCL kernels are compiled at runtime, for reasons of portability despite the additional performance overhead. To do so, the programmer needs to load the sources of the kernel using *LoadOpenCLKernel* and create the associated program using *clCreateProgramWithSource*. Then the program is built (*clBuildProgram*) and the kernel is created (*clCreateKernel*).

The last phase of the execution is about the organization of the work-groups and the distribution of the workload, shown in figure 3. First the programmer needs to allocate the device side buffers and associate them to the hots side ones (*clCreateBuffer*). Then, these device side buffers are mapped to the kernel arguments (*clSetKernelArg*). The programmer then needs to specify the number of work-items with the *globalWorkSize*, and the way they are organized in work-groups using *localWorkSize*. Then the kernel is submitted for the given configuration using *clEnqueueNDRangeKernel*, and finally the result is read back to host memory using *clEnqueueReadBuffer*.

```
// Create the input and output arrays in device memory for our calculation
d_A = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, mem_size_A, h_A, &err);
// Same for B and C

// Launch OpenCL kernel
size_t localWorkSize[2], globalWorkSize[2];

int wA = WA;
int wC = WC;
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&wA);
err |= clSetKernelArg(kernel, 4, sizeof(int), (void *)&wC);

localWorkSize[0] = 16;
localWorkSize[1] = 16;
globalWorkSize[0] = 1024;
globalWorkSize[1] = 1024;

err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);

// Retrieve result from device
clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, mem_size_C, h_C, 0, NULL, NULL);
```

Listing 3: C-based pseudo-code of matrix multiplication in OpenCL, device code execution.

The code of the OpenCL kernel for this example of matrix multiplication is displayed in appendix A on listing 21.

3.1.2 High-Level Heterogeneous Programming Environments

In the previous section, we presented CUDA and OpenCL, two frameworks that ease the accessibility of devices in the context of general purpose computing. The frameworks provide the means to accomplish the heterogeneous execution, but there is a large amount of management remaining to the programmer. The synchronization

between the host and device memory, the synchronization of the device side code, and the workload mapping (associating certain parts of the workload to the accelerator) happen explicitly by the developer as shown previously on the matrix multiplication example.

The approaches in this section provide additional functionalities on the aforementioned mechanisms. On the host side, most of them provide a task based programming model implemented on top of threads. In that context, a *task* is a well defined piece of work that performs a set of operations on some data. Focusing on the task level, the developer does not need to manage explicitly the threads.

3.1.2.1 OpenACC - An Industrial Specification For Heterogeneous Computing

OpenACC [126] is a directive-based industrial standard. It was originally proposed as a fork of OpenMP aiming to ease the programmability of accelerators, targeting GPUs. The underlying memory model is based on separate host and device memories, without automatic synchronization. The programmer needs to be aware of the different memories, and orchestrate data movements using clauses that specify the direction of data movement (*copyin*/*copyout*). OpenACC implementations come with proprietary compilers able to generate GPU-compatible binary. The notion of the gang, or vector length is defined corresponding to the notion of work-groups we met in OpenCL. The most significant directives are the *parallel*, *kernel* and *loop* ones, to annotate parallel regions, computationally intensive code areas and parallel loops respectively.

3.1.2.2 Runtime Support For Heterogeneous Task Execution Using Charm++

Charm++ [68] was originally introduced as a C++ API targeting parallel platforms relying on the Charm parallel programming system [102]. It was created to overcome the bottlenecks imposed by concurrency within parallel computing, involving load scheduling, balancing and synchronization, satisfying the definition of a runtime system in our context.

Charm++ is a portable platform (that can be deployed on any shared memory homogeneous multicore machine) responsible for the orchestration of the application building blocks that are called *chares*. A chare is similar to a task. It is an individual work-unit that can communicate with its environment (other chares) via messages or through shared memory. Chares are dynamically distributed over the computational

resources in a balanced fashion, ensuring a better resources utilization. Every call to the runtime system and message exchange is asynchronous. CHARM++also provides several data structures that are tagged according to their access permissions, including read-only objects, write-once objects, accumulators and distributed tables.

Multiple scheduling policies have been implemented in CHARM++ in a modular way. Among them one can find a random one (chares are randomly distributed to the processing units), an adaptive contracting within neighborhood (dynamically created chares are scheduled to processing units near the processor executing the parent chare) and a central manager (where the scheduling of that chares is happening from a centralized component that has an overview of the system).

The programming environment exposed to the programmer with CHARM++ is composed by a source to source compiler (translator interface) as well as the Charm runtime system. The translator transforms the CHARM++ constructs into C++ ones. The Charm runtime system is written in C. Every additional functionality, feature or strategy (queue and memory management, load balancing) can be written separately and linked as a module to the core of the system.

In the context of heterogeneous computing, when GPUs got widely adopted in high performance computing, CHARM++ was extended in order to incorporate GPUs as supplementary general purpose processing units.

Wesolowsky et al. [125] extended CHARM++to provide GPU support. HYBRIDAPI is the CHARM++ component responsible for the GPU integration, and relies on CUDA to communicate with GPU devices. A work queue is associated to every GPU connected to the platform, holding the kernels assigned to the device. The user needs to declare the access rights of every kernel to every data structure it is associated with that will later run on the GPU, as well as a callback function that will be invoked by the GPU once the work request is completed. At runtime, the scheduler will push the kernel in the appropriate GPU queue, and ping for the completion of its execution in order to satisfy the dependencies of the rest of the chares or kernels. In order to minimize the overhead of data exchange, data are sent back to the host along with the execution of the next kernel of the queue, overlapping execution and communication when possible.

In [104], Robson et al. extended this work, focusing on the generation of the GPU code and the dynamic target selection. The GPU manager presented in [125], was inherently asynchronous, managing transparently blocking operations like memory allocation on the GPU side. Robson et. al. utilize ACCEL, which is a framework that automatically extracts kernels that are annotated with the keyword `accel` by the

user to be executed in a GPU, managing automatically the callback function necessary to the GPU manager. ACCEL also handles data transfers between the CPUs and the GPUs, inheriting and augmenting the computation and communication overlapping when possible.

G-Charm [122] is another effort to augment the capabilities of HybridAPI, in order to have a better utilization of the GPU resources. G-Charm focuses on dynamic heterogeneous scheduling, automatic memory management with emphasis on data transfer minimization and work agglomeration merging multiple chares to larger kernels when this is beneficial.

3.1.2.3 OmpSs - Runtime Support Of FPGA For SoC

OmpSs [36] is a directive based environment for the execution of task-based applications on heterogeneous platforms. It provides a task-based programming model, as well as a runtime system responsible for the dependency resolution, and the orchestration of execution of an application on platforms with a series of devices connected with a multicore CPU.

In the scope of the programming model, every accelerator is considered a processing unit that can efficiently execute parts of the application. In the introduction of OpenMP (chapter 2) the definition of tasks was introduced, as code blocks that are associated to some data and can execute in parallel. It is the same definition that applies to the notion of a task in OmpSs. The developer needs to annotate a code block (usually a function) as a task, and to indicate its associated data using the *input, output* constructs to indicate the nature of the dependency. In case the task can be executed BY one of the available devices, it needs to also annotated, specifying the device target, and forcing the copy of the dependencies. If no target is explicitly specified, the task must be executed in one of the available CPU cores.

The OmpSs programming model is implemented with a dedicated compiler and the Nanos++ runtime system. Nanos++ creates a pool of software threads, that correspond to workers, and are associated to the computational units (including both the CPU cores and the available devices). Scheduling policies can be implemented as plug-ins, indicating the principle upon which tasks will be assigned to workers during the execution.

3.1.2.4 Anthill - Runtime Support For Large Heterogeneous Distributed Environments

Anthill [40] was introduced as a runtime framework for heterogeneous distributing computing, originally designed for data-mining applications. Such applications are made of a series of operations performed on data chunks that are distributed all over the system. The computations performed on the data are split into individual stages, called in this context *filters*, that are pipelined to decrease the total execution time. Those filters are typically organized in cyclic graphs, that are concurrently applied on the data chunks that are distributed all over the system. Filters are further decomposed into tasks, that may be executed on different nodes of the platform. The motivation behind the runtime design of Anthill was to provide the infrastructure that exploits both the task and the data parallelism as well as the asynchrony of the nature of the algorithm.

In order to achieve the aforementioned goal, Anthill provides labeled stream support, a global persistent storage structure and a termination detection mechanism. Labels are applied, in order to ease the dynamic routing of the exchanged data, that are streamed between tasks. The global storage refers to a concept that the internal state of the filters are stored and propagated to multiple nodes, along with the associated data chunk, in order to decrease the distance between data and computing resources, as well as to provide a certain level of fault tolerance. The termination detection mechanism refers to applications where the graph is cyclic, and a loop counter is kept internally during the execution.

In [117], Teodoro et al. present a new event-oriented model for Anthill, targeting a more efficient utilization of the resources at the node level. This new event-based approach enables a transparent and automatic handling of data dependencies. The programmer can focus on the processing that should occur on the arrival of new data of a stream. The data exchange is no longer managed in a point to point fashion between the filters, but there is an event controller, orchestrating the message exchange interacting with a centralized communication module and event handlers associated to filters. Monitoring the communication module and the events produced by the event handlers of the filters, scheduling policies can be ported and applied for better resource utilization.

In [118], Anthill was extended to exploit GPUs as supplementary processing resources. It is the event handler that needs to be extended, in order to permit the data streams to be offloaded on a different kind of resource. The communication with the GPU is done with CUDA. With the event-based model, dependency resolution

occurs on the fly, and the runtime system can choose the processing unit that will process a certain filter based on the event handlers provided with it. In terms of scheduling, two scheduling policies are implemented. The first one is *first come First served*, where the first available resource will process the next pending event that entered first in the events queue, The second one is *weighted round robin*, where a weight is dynamically assigned to every handler of every event, and the first available resource will serve the heaviest weighted event with the appropriate handler.

Teodoro et al. claim that any device can be supported by the framework, mentioning the Cell co-processor and FPGA specifically, but to the best of our knowledge, no FPGA support was added so far.

3.1.2.5 StarPU - A Runtime System For Heterogeneous Platforms

StarPU is a runtime environment providing an API that is used as a library to exploit parallelism in heterogeneous machines. Parallelism in StarPU is expressed through the notion of tasks; autonomous chunks of work with input dependencies, that can be executed on any of the available underlying processing units. A task is abstracted using the notion of a codelet, which holds the information of its dependencies, along with its available implementations. An application is then expressed as a task graph.

As a runtime system, StarPU provides the necessary mechanisms to detect task dependencies, handle task communications accordingly and provide scheduling mechanisms for an efficient workload mapping. The reference platform architecture targeted by StarPU is the one of figure 2.2, on top of which it implements a relaxed consistency shared memory model. To do so, StarPU employs a MESI like protocol, to update every private memory of the platform with the valid data. The programmer does not need to explicitly orchestrate the data movement, since this is done transparently by the runtime. On the scheduling part, there is the StarPU scheduler responsible to assign tasks with resolved dependencies to processing units. This gives the flexibility to programmers to chose among different scheduling policies provided, or implement their own, according to the characteristics of their application.

StarPU is one of the main building blocks of our framework and will be extensively described in chapter 4.

3.1.2.6 Harmony- Runtime Support For Heterogeneous Many Core Systems

Harmony [32] was introduced as a programming model and an execution environment for heterogeneous platforms. The reference architecture for the environment is

the one of an heterogeneous node, where multiple CPU cores can access the same memory under a global address space and accelerators can be attached usually via a PCIe bus. The architecture demonstrated in [32] was based on an heterogeneous node assembled by CPUs and GPUs, while a speculative example was provided to a platform that could include an FPGA.

Diamos et al. [32] suggested a programming model based on three elements: the computational kernels, control decisions and a global memory address space. Computational kernels are well defined code segments, associated with some data (analogous to the task concept in our context). Every kernel can be executed on processing units of different architectures, as long as the programmer provides an implementation for every such. Within the description of computational kernels, the input and output data need to be annotated, allowing a dependency-driven scheduling approach. The shared address space creates the need for the management of coherence and consistency issues that occur with the concurrent access of different kernels to some shared chunks of memory, as well as for the management of data transfers between the main memory and the private memories of the accelerators, managed by the runtime.

Harmony was extended to support Kernel Level Speculation (KLS) [31], which is speculative execution of kernels, similar to the one that is employed at the thread level transparently to the programmer. Since the input/output of every kernel is declared by the programmer, data dependency graphs can be created dynamically.Speculation is employed in order to minimize the overhead of blocking control decisions. In order to allow this speculative kernel execution, several mechanisms are employed; one for variable renaming (similar to register renaming that is happening in hardware level), one for resources de-allocation and control flow reset on misspeculated scenarios.

3.1.2.7 Qilin - Adaptive Workload Mapping For Heterogeneous Machines With CPUs And GPUs

Qilin [84] is among the first frameworks that demonstrated the importance of dynamic load balancing in platforms with heterogeneous computing resources, which in the context of this approach is called *adaptive mapping* (similar to the notion of scheduling according to the terminology we follow within our work). Luk et al. demonstrated that executing a simple matrix multiplication on a platform with a multicore CPU and a GPU can deliver different results based on the amount of computations assigned to every processing unit. Furthermore, in the same study they demonstrated that the optimum load balance varies for different problem sizes of the same application.

Qilin provides a C/C++ API, a dynamic compiler that creates machine code for both the CPU and the GPU, development and monitoring tools as well as a scheduler. The dynamic nature of the compiler is expressed by its ability to identify the device code, and use NVCC for its compilation. The API is based on top of Intel Thread Building Blocks (TBB [73]) for the management of parallelism on the CPU cores, and Nvidia CUDA for accessing the GPU.

For the adaptive mapping, Qilin uses a history-based approach for the estimation of the workload instead of an analytical performance model that would have been obtained from static analysis. Every time Qilin executes an application for the first time, a training run is performed, offloading parts of the workload to the underlying processing units and registering the obtained performance to a database. For every other execution of the same application, the workload mapping happens based on the results obtained from the training run. To the best of our knowledge, Qilin was not extended to support other accelerators rather than the GPU.

3.2 FPGA Accessibility From The Hardware Community

This section introduces the FPGA integration approaches from the hardware community. Most of the approaches focus on platforms with only CPUs and FPGAs. The main focus is to abstract as much as possible hardware side so that software and hardware can co-operate.

In our presentation we have distinguished three integration levels; integration on the thread level (section 3.2.1), the task level (section 3.2.2) and the process level (section 3.2.3). The criterion we based our taxonomy on is how the accelerated parts of an application communicates with the rest of the environment.

3.2.1 Thread Based Approaches

This section presents some of the most popular thread based approaches of integration of FPGA in a heterogeneous environment. Conceptually a thread is a set of instructions that are executed sequentially. Threads can be executed concurrently and independently, sharing the same memory space.

As an access point, threads is a versatile level, that can deliver increased performance. Nevertheless, to ensure the correctness of the execution, developers need to deal with low level mechanisms such as thread communication, synchronization and scheduling.

In software development, there is a long history in thread based programming. The POSIX library is a well established API in C/C++ that provides support for the basic functionalities of a thread, on top of which the heterogeneous frameworks of section 3.1 have been developed. The majority of the approaches of the current section attempt to provide POSIX compliant interfaces, where hardware threads should be able to communicate transparently with their software counterparts.

3.2.1.1 HThreads - Hybrid Threads; A POSIX Compliant Thread Based Abstraction For Reconfigurable Computing

HThreads [15] is a design flow opting to provide a unified thread based model for architectures with reconfigurable components, abstracting the underlying hardware. It was introduced by the time FPGA boards became capable enough to be coupled with multiprocessor systems on chip. From the application point of view, the model corresponds to a fork-join execution, when a parent thread creates a number of children threads that can be executed transparently on the underlying computational resources and then reunite to the single parent thread gathering the result of the computation. The introduced design flow, along with the runtime support provided by the framework, enables the programming of an heterogeneous MPSoC using standard POSIX-compatible programming abstractions, by using a hardware-based microkernel to provide the OS back-end to the rest of the components assembling the system.

The compilation and synthesis flow starts from a C application that is processed by the HybridThreadsCompiler (HTC) (yet another High Level Synthesis tool), that is a concatenation of two tools, HIFGEN and HIF2VHDL. HIFGEN is a GCC based compiler that creates an intermediate representation for the hardware targets called HIF. HIF2VHDL is the tool that performs the synthesis of the HIF sources to VHDL.

As with the other thread based approaches like ReconOS (3.2.1.4) and SPREAD (3.2.1.2), HThreads delivers hardware mechanisms that enable the transparent integration of the hardware threads to the rest of the system. These mechanisms include a component that performs basic scheduling, a component providing synchronization (mutexes) support, as well as an interrupt handling unit for the hardware threads. The design decision to migrate this component on hardware was taken with performance being the basic criterion. In addition, a hardware thread interface (HWTI) is added along with every hardware thread (similar to the HTI shown in section 3.2.1.2) that allows the communication between threads as well as between a hardware thread and the OS. A distinctive feature of this HWTI, is a local memory

that is coupled to every HWTI instance, and is globally accessible, that allows fast communication without contention/congestion on the communication bus.

In [8] the platform is evaluated on a set of application including parallel π approximation, divide and conquer parallelization and mailbox-based communication.

3.2.1.2 SPREAD - A Thread Based Approach For Streaming Applications

SPREAD [124] is a streaming oriented programming environment for architectures with reconfigurable components. It is a thread based approach where tasks executed in hardware are encapsulated into threads, resembling the POSIX paradigm of the software counterparts. The application is decomposed into tasks, that are further manually organized into threads.

Since the focus is put on streaming application, they consider three basic operations performed by a thread, which is getting the data associated to the computation, the computation phase itself, and then providing the data to the next consumer. To facilitate switching between hardware and software, the model is based on a switchable thread approach (similar to software hardware codesign), where a software implementation of a thread is provided supplementary to every hardware one. Moreover, for every hardware thread, a software delegate is created, called the stub thread, responsible for the monitoring of the hardware thread on the software side. A Hardware Thread Interface (HTI) is coupled with every hardware thread enabling the communication between hardware and software threads as well as hardware threads between each other.

For the management of the reconfigurable resources, a dedicated component is employed in order to detect idle hardware threads, and replace them without manual intervention with others. The resource manager is designed so that partial dynamic reconfiguration techniques can be employed in the future. Along with the resource manager, a hardware thread manager is associated to hardware threads responsible for their creation and termination.

There are three different cases of inter-thread communication depending of whether the communicating components are executed on hardware or software. In case both threads are implemented in software, the communication will by delivered using memcpy(). In case there is a software and a hardware communicating thread, they communicating via a synchronous point to point connection using DMA. In the last case, where both threads are executed on hardware, they communicate over FIFOs synchronized by the HTI.

3.2.1.3 FUSE - Front-End USEr Framework To Utilize Hardware Accelerators Under An OS Abstraction

FUSE [64] is another approach that eases the programmability of reconfigurable hardware accelerators. The reference model for an application developed using FUSE is thread based. Threads are assembled out of tasks, that can be executed either on software or on hardware. FUSE threads are POSIX compliant, and are integrated into the PetaLinux OS.

In order to provide a transparent abstraction for task execution, FUSE relies in a user space and a kernel space component respectively called the Top Level FUSE Component (TLFC) and the Low Level FUSE Component (LLFC).

TLFC is provided as a library to the user space. According to the model, every task on the TLFC layer is associated to a context, that curries information about its state. TLFC provides the mechanisms to create, initialize, execute and destroy contexts. It also provides a set of functions that allow the communication with the LLFC and with the OS, allowing the dynamic OS module loading and unloading that permits the actual task execution on hardware. TLFC is also the component responsible for tasks scheduling. According to the literature, FUSE provides the flexibility for any scheduling policy to be implemented and applied on a given application, although it comes with a default scheduling policy which is the following. If a task has a hardware counterpart instantiated in hardware and that counterpart is idle (not executing another similar task), then TLFC and LLFC are orchestrating the execution in the hardware component. Otherwise, if there is no hardware implementation, or the available instances are busy, the task is executed in software.

LLFC is the kernel side counterpart, responsible for the communication, the synchronization and the monitoring of the hardware tasks. Every hardware task instance is coupled with a hardware accelerator interface, which is the connector between the task and the LLFC. More specifically, for every such interface, a loadable kernel module is created dynamically during the runtime while the kernel is executing, within the LLFC layer.

The platform is evaluated on three applications that are a 2D-DCT transformation, a 3DES encryption and a SOBEL filter. The evaluation is based on the execution time per task, when this iss executed in software and hardware, with and without taking into account the time overhead associated to the loading of the kernel module of every hardware task, showing that there are cases where the hardware tasks significantly outperformed the software ones.

3.2.1.4 ReconOS- Multithreaded Programming Environment For Computers With Reconfigurable Components

ReconOS is an operating system for computer architectures that incorporate reconfigurable hardware components. The framework was originally developed for SoC architectures; CPU cores coupled with an FPGA on the same die.

The underlying model relies on threads, that need to be explicitly managed by the application programmer. More precisely, it is the application developer who is responsible for decomposing the application into individual threads, managing their synchronization (through semaphores and mutexes) and their communication (through shared memory or message exchanges for the communication between the device and the host). From the programmability point of view, the API provided by ReconOS is very close to the POSIX one, with calls for semaphore handling, mutexes, condition variables, mailboxes, and thread orchestration being almost identical.

ReconOS software threads are identical to the POSIX ones, and can be implemented using the POSIX library. From the hardware point of view, the massive concurrency of hardware designs requires a synchronization mechanisms for the software side compliance. For this to be accomplished, every hardware kernel, is coupled with another hardware component, that is written in VHDL and is implementing the hardware thread synchronization mechanisms. Thread creation and termination is fully POSIX compliant.

In order to make the resource sharing feasible, an indexing mechanism is introduced for the hardware threads. Threads memory is shared, leaving the consistency and coherence hazards between software and hardware threads to be handled by the programmer.

In [106] is a demonstration of utilizating Reconos for an image processing application.

3.2.2 Dataflow Based Approaches

In this section we present a number of approaches, that target applications written using a dataflow model and that try to provide a model level abstraction of the FPGA. According to this model a program is composed by concurrently executing entities, called actors, communicating through explicit channels. Actors have dynamically created input dependencies and will remain in blocking state until they are satisfied. The resulting application then takes the form of a static dataflow graph, that is the graph is known at compile time and does not change during execution. The goal of

this level of abstraction is to automatically manage the communication between the device and the host.

3.2.2.1 FOSFOR - A Model For Dataflow Applications On Architectures With Reconfigurable Components

FOSFOR [52] is a framework that provides a transparent abstraction layer for applications following the Synchronous Data Flow model, deployed on System on Chip architectures. Synchronous Data Flow model derives as a restriction of the data flow model. Here actors produce and consume a fixed amount of tokens on every channel which is known at compile time. Within the project, the notion of Dynamic Hardware Actors is introduced, corresponding to tasks that are executed in hardware that can be configured dynamically during the runtime.

The design flow involves multiple levels of synthesis from the Synchronous Data Flow description of the application to the actual execution. The underlying platform is composed by a combination of multiple softcores - hardware designs of CPU cores configured on the FPGA - (MicroBlaze) and a number of reconfigurable hardware components. Starting from the Synchronous Data Flow description, an architectural mapping is required in order to associate the actors to the processing units that they will be executed on. Then it is the implementation of the actors, which is conducted using C and C++ for the software and a hardware-level description (using VHDL) for the hardware ones. The result of synthesis is a hardware design which is then used to configure the FPGA and perform the actual computation.

Once the amount of softcore processors is determined taking into account the amount of the available resources, the remaining area on the chip is split into reconfigurable partitions where the hardware actors will be executed on. Every hardware actor is represented by a hardware thread in the OS level, while a software actor is associated to an OS one. For the orchestration of the reconfigurable areas, Flexible OS is chosen as a hardware operating system. For the OS management of the software part RTEMS OS is employed.

Hardware actors are represented by hardware threads at the system level. Hardware threads are composed by a static component, that is responsible for the communication with the hardware OS as well as for the communication with the network, and a dynamic one that contains the hardware actor (task) as well as some associated control and a private memory to facilitate the communication through the network interface.

In order to overcome the complexity of the communication that comes because of the different nature of the processing units, a middleware layer is established between the application and the OS of the software and the hardware units. This layer provides a virtualization (according to their terminology) of the resources, providing clean interface for the thread communication called Virtual Channels.

The framework is evaluated on an image processing application that performs pattern recognition and tracking on an infrared video stream. The evaluation refers mostly to the resources allocated to every component on the system, not to the performance.

3.2.2.2 ReConfigMe - An Operating System For Reconfigurable Computing

ReConfigMe [127] is another approach to provide a unified OS abstraction for computing systems with reconfigurable hardware components. It is designed distinguishing three different entities, the user tier, the OS tier and the platform tier that are connected over a TCP/IP network. The reference hardware architecture is a conventional multicore processor coupled with an FPGA via PCIe, that communicates with the OS and the user client via a TCP/IP network.

The reference model of an applications designed for the platform is the one of a flow graph, where nodes correspond to processes. The application graph should be designed so that computation will start from a source node that contains the source data in the beginning of the execution, and it will finish at an exiting node connected to a data sink where the result data will be written on. Communication between processes is performed through the board memory, so every hardware implementation of a node is coupled with a memory interface. The placement and routing of every node, and consequently of the entire design is performed statically before the execution.

The platform tier is the one closest to the hardware among the 3 tiers mentioned previously. Its main objective is to provide control of the underlying architecture to a remotely connected user. A Hardware Abstraction layer (HAL) is provided to allow user read and write bitstreams to the FGPA, as well as to communicate with the executed processes via the onboard memory.

The OS tier is also named the Colonel. It is a platform where the user can remotely connect and create the bitstream corresponding to the application. Colonel ensures that the user design satisfies the spatial restrictions imposed by the FPGA board. It directly communicates with HAL, serving as its client side, providing the bitstreams synthesized by the user. Last but not least Colonel performs the memory management

required on the hardware side for the application data provided by the user. A part of the OS tier is the management of the user client, making sure that application data can be passed through a secure connection abstracting the underlying communication protocol.

It is the user tier that allows developers to deploy their applications. It provides a basic API to communicate with the Colonel, as well as to stream data to the platforms memory.

The proof of concept is made using a blob tracking application used in image processing. The evaluation mostly concerns the hardware resources required for the deployment of the application.

3.2.2.3 FlexTiles- Flexible Tiles For An Heterogeneous Manycore Architecture

FlexTiles [66] is a project introducing a programming model for heterogeneous many-core architectures that include general purpose processors, Digital Signal Processors and embedded FPGA connected over a network on chip. The hardware resources are organized into individual entities called tiles with a varying granularity (from a single general purpose processor to groups of resources). An application adjusted to the FlexTilesmodel is decomposed to actors, that are later on mapped to and executed by the processing units of their associated tiles.

According the FlexTiles model, software is organized in layers. It is the CoMiK microkernel that provides a primitive level of scheduling and memory management, on top of which the operating system POSE provides a second level of scheduling along with the infrastructure for task communication. Scheduling is conducted in two levels, one closer to the tiles and one from a higher point of view, by a virtualization layer, that is communicating with resource managers extracting runtime information about the state of the tiles.

3.2.3 Process Based Approaches

In this category we present approaches on the process level. Here the parts of the program that are executed on the FPGA have the form of Linux processes, communicating through pipes. Usually a hardware process has a software counterpart, to communicate with the rest of the software processes making the application. This software counterpart is responsible for the synchronization of the process and the communication between the host and the device.

3.2.3.1 BORPH - Berkeley Operating System For ReProgrammable Hardware

BORPH [111], as the acronym mandates, provides an Operating System level support for platforms with a various number of reprogrammable computing resources, by extending the standard Linux Kernel. The main motivation was to enhance the accessibility of such platforms, designing a system that can exploit the flexibility of those resources, endorsing their ability to be partially reconfigured dynamically.

A program in BORPH corresponds to a number of processes, with hardware processes being the equivalent of the software ones, corresponding to the part of the program that is executed on the reconfigurable fabric. Technically a hardware process in BORPH is an executable binary object file, that contains information about the reconfigurable resources like the configuration (hardware design) of the process itself.

The framework is made of a kernel module and a user API that provides a set of system calls to the module. The module is responsible for the request handling that mainly correspond to the allocation and configuration of the hardware resources. The API includes a series of system calls that implement a message passing interface allowing a hardware process to access data files or to communicate with other processes through pipes. In order to enable the communication between software and hardware through file accesses, the traditional file system was extended to IOREG [4], where files are created dynamically for every configured hardware process.

3.2.3.2 SPORE- Simple Parallel Platform For Reconfigurable Environment

SPORE [48] is an execution model designed for parallel computing on platforms that include reconfigurable hardware. In [46] Foucher et al. present their application model, according to which an application is divided in kernels, communicating using MPI. Kernels along with their dependencies are described in XML.

The SPORE model is built around three principles: execution-communication decoupling, online codesign and virtualization (following their terminology). Execution and communication decoupling refers to the concept that the communication interface should remain the same regardless of the underlying processing unit. Online codesign refers to the concept that every kernel should come with a software and a hardware implementation. Virtualization refers to the concept that multiple hardware implementations of a kernel can be provided, leaving the decision of where a kernel will execute on will be taken at runtime depending on the availability of the

underlying resources, hence in the application description a kernel is represented by a virtual abstraction.

The model comes with two implementations, the Software HPC Platform (SHP) responsible for the software orchestration, and the Hardware Stream Dynamic Platform (HSDP) responsible for the execution of decisions coming from the concept of virtualization.

From the architectural point of view, the entry point is a node. For their evaluation they used a Virtex 5 FPGA coupled with a PowerPC 440 CPU representing their node reference. The operating system is installed on the CPU core, playing the role of the host that is responsible for the internode communication as well as the intranode orchestration. Their processing nodes are multiple MicroBlaze softcores, running pure MPI processes, communicating with the host through mailboxes -bidirectional FIFOs on top of shared memory enabling the message passing. In the SPORE approach the global scheduler is physically detached from the computing nodes, and scheduling decisions are communicated to local schedulers that are running to every node host. The same applies for the data server, that can also be accessed through the network, and has a host counterpart called storage manager.

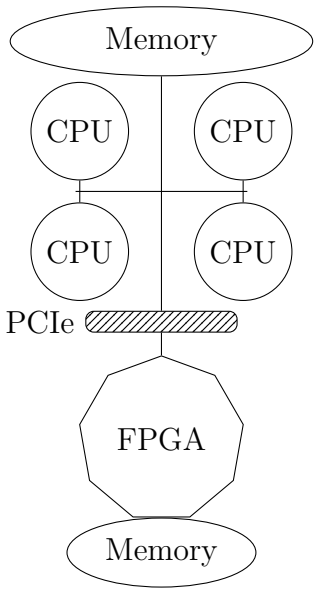
3.3 Bridging The Gap : Heterogeneous Programming With FPGA

This section presents the number of approaches that have attempted to provide accessibility of FPGAs in the scope of a truly heterogeneous environment including any type of processing units such as CPUs, GPUs and FPGAs. Approaches here, have already been presented in the previous sections. In this section we present how they were extended to support FPGA. We first present how OpenCL and CUDA were extended to support FPGA. Then we introduce extensions to OpenACC and OmpSs, two frameworks that incorporate FPGA on a higher level than CUDA and OpenCL.

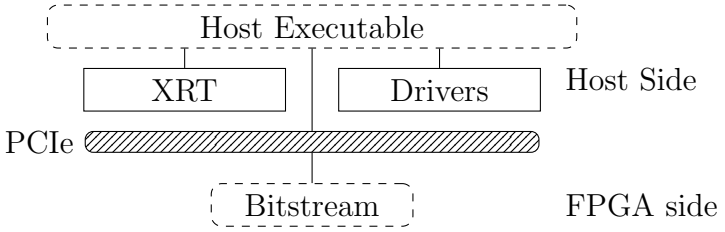
3.3.1 OpenCL Support For FPGA

OpenCL as an open standard targeting the accessibility of accelerators for general purpose computing is currently supported by the two major FPGA vendors, that is, Xilinx and Altera. SDAccel is a software environment from Xilinx while Intel FPGA SDK for OpenCL is the equivalent development kit from Altera that provide OpenCL support for FPGA. We will use as a reference SDAccel, to present the structure of an application with OpenCL kernels executed in FPGA.

Figure 3.2 provides an overview of the reference architecture as well as the software layout of an application with OpenCL kernels targeting FPGA using SDAccel from Xilinx. The host code along with the Xilinx runtime and the required drivers are running on the CPU nodes, while the OpenCL kernels are configured on the hardware fabric. XRT manages transparently the low level mechanisms of the data exchange, and delivers some basic scheduling mechanisms of the application threads.



(a) Reference architecture.



(b) Software layout.

Figure 3.2: SDAccel reference architecture and software layout. The reference architecture is similar to our heterogeneous reference architecture. A series of CPU cores connected with a FPGA over a PCIe bus. The FPGA is configured using a bitstream. The host executable is linked with the Linux drivers and the runtime system of Xilinx (XRT).

OpenCL is so far the most widely adapted framework to utilize FPGA on the industrial scale level, with companies like Amazon enhancing their cloud infrastructure with such devices [12]. On the overview of example applications provided in the developers page by Amazon [13], we can see that the majority of the application were developed using OpenCL for the accelerated kernels, and SDAccel as a framework to deploy the application across the resources. The Amazon paradigm is distinctive since resources are accessible by the general public. A user can allocate special resources, called *instances*, that may contain FPGA as accelerators. The instance sizes may differ, from 8 virtual CPU cores coupled with one FPGA, up to 64 virtual CPU cores coupled with 8 FPGA devices. The access to the platform is not free, although

a wide range of code generation, hardware design synthesis and FPGA configuration tools are provided with the instances.

3.3.2 FCUDA

CUDA [91] was introduced in section 3.1.1 as a general-purpose parallel programming model for the graphics processors of Nvidia. A CUDA program consists of one or more phases that are executed on either the host (CPU) or a NVIDIA GPU device in a fork-join strategy. The phases that exhibit little or no data parallelism are implemented in the host code, while phases exposing rich data parallelism are implemented in the device code. The host code is ordinary C/C++ code, while the device code is an extended ANSI C with keywords for labeling data-parallel functions, called kernels, and their associated data structures. The parallelism is expressed by a hierarchy of thread groups called blocks. In CUDA, the host and the devices have separate address spaces. CUDA supports a number of features such as asynchronous execution, concurrency with streams, event monitoring, unified virtual address space (UVA), and multi-device support.

FCUDA [97, 98] is a framework that allows CUDA code with some supplementary annotations to be synthesized in hardware and further executed on a FPGA.

The synthesis flow of FCUDA is composed of 4 steps, from the original C/CUDA code to the final hardware description and then the final hardware design, demonstrated in figure 3.3.

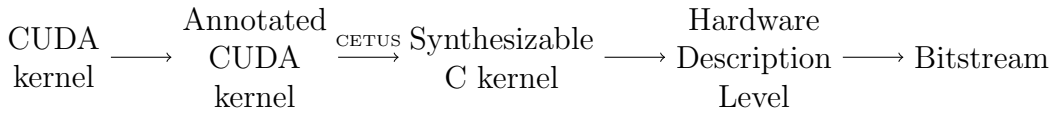


Figure 3.3: Synthesis flow of CUDA kernels for FPGAs using FCUDA. The synthesis starts from a vanilla CUDA kernel, that is annotated with pragmas compiled into a synthesizable C kernel that is later translated into hardware description level and then to a bitstream.

The first layer corresponds to some code annotation with pragmas, used by the programmer, that allow the synchronization of the kernels and some irregular control flow statements (like breaks and returns). The annotated code is then parsed by Cetus [75], a source to source compiler than performs the translation and optimization of the enhanced C/CUDA code to some synthesizable C code. The code is then processed by Vivado HLS, for the hardware-level description of the design, and then

by VIVADO, the back-end from Xilinx that will finally deliver the hardware design that corresponds to the CUDA code.

On the runtime aspect, FCUDA and OpenCL both provide mechanisms to access the device and feed the kernels with the corresponding data, but the dependency analysis, the decisions of which data need to be moved, and the scheduling of the tasks/kernels remains to be handled by the programmer manually.

3.3.3 OpenACC Support For FPGA

In [76] Lee et al. proposed a framework that enables the execution of an OPENACC program on an FPGA-enhanced platform. The core of the work is based on OPENARC, a source-to-source compiler used to convert and optimize OPENACC code to OpenCL for the FPGA. The AlteraBackend OpenCL compiler was used in order to create the hardware design, that was targeting an Altera Stratix V FPGA. The approach was evaluated in a series of kernels from different benchmark suits (OpenARC suit, Rodinia, Altera suit). Performance results correspond to a comparison between the execution times of different configurations on different architectures, without any truly heterogeneous scenarios. Meaning, for every execution of the applications, the compute intensive parts were executed either on an FPGA or on a GPU or on a Xeon Phi, but there were no results combining different resources.

3.3.4 OmpSs Support For FPGA

In [42], OMPSS is extended to support FPGA on an all-programmable SoC board. To the best of our knowledge, OMPSS is not further extended to support more scalable architectures, where FPGA are connected as accelerators to bigger nodes.

For the code generation, Mercurium, a source-to-source compiler, is employed to handle the FPGA related tasks. For every FPGA task, mercurium produces a C file with the body of the task annotated with pragmas, that is furtherly processed by Vivado HLS and VIVADO in order to obtain the hardware design corresponding to the FPGA tasks. Along with the C-sources that are further processed by the FPGA tool-suit, Mercurium generates another C-file associated to every task, that works as the software counter-part of the task. This other file incorporates the calls to the runtime as well as calls for the data exchange to and from the FPGA device.

For the runtime management the NANOS++ runtime system is employed. The scheduler of NANOS++ is extended to allow multiple task submission to the FPGA. This is due to the fact the amount of CPU cores is limited, and the thread that

performs the task submission can not block waiting to receive back data, leaving part of the hardware resources idle. To allow multiple tasks submission, double buffering and pipelining features are employed.

For the evaluation of the approach, experiments are performed on a series of application including matrix multiplication of two granularities, Cholesky factorization and Covariance computation. The comparison is between the execution time running on a single ARM core, versus the execution running on one or two FPGA tasks. Results indicated that the execution of the the FPGA outperformed the single ARM core, in every case, while the heterogeneous scenarios where tasks are executing both on an FPGA and on the CPU core, did not show further performance improvement.

3.4 Discussion

In sections 3.1 and 3.2 we presented a plethora of works that aim to ease the accessibility of FPGA as well as to provide programming environments that can exploit the parallelism of the underlying platforms without imposing a big overhead. The amount of approaches, as well as big recent industrial efforts like the oneAPI project by Intel [63], expose the need for a unified programming environment to exploit the processing power of massively heterogeneous processing machines, including reprogrammable hardware components.

The approaches from the hardware community can be categorized in three groups, those who provide a thread-level, those who provide a task-level and those who provide a process level integration of FPGA. Approaches of the first group associate a FPGA with a software thread. Most of the times, mechanisms for the thread creation and the device accessing are provided by the framework. Although thread synchronization and device memory synchronization need to be managed explicitly by the programmer. The approaches of the last two categories, suffer either because the reference application model is too restrictive, or because the abstraction layer is not close to what software developers expect. Moreover, the reference platform architecture for the majority of the related work of this category, is the one of a single FPGA coupled with a small number of CPU cores, which imposes scalability and portability barriers to their integration in a modern high performance computing system.

On the other hand, software community offers a wide range of environments where parallelism can be expressed in a manageable way, without the difficulties of manual thread manipulation. The building blocks of those platforms are tasks, that come at a finer granularity than threads. The memory synchronization, the resolution of

the dependencies, the data exchanges, and the workload mapping are handled by the runtime system.

OpenCL so far is the most widely adopted accessibility framework. It was originally created to ease the accessibility of GPUs, but it evolved successfully to support FPGAs, to the point that FPGA vendors nowadays have integrated OpenCL support in their tool-suits. Although, despite the fact that OpenCL provides a rich environment for the accessing of the device and the data exchange, the resolution of the dependencies and the data exchange, as well as the workload scheduling are not completely automated.

The approach of OmpSs (3.1.2.3) is the closer to the one we have chosen. OmpSs provide runtime system support, for parallel applications exposing task parallelism, handling automatically data dependencies and data exchange. Although the FPGA integration is bound to SoC architectures. This impose a limitation regarding the scalability of the approach, since the underlying architecture is bound to a single FPGA and a small number of integrated CPU cores.

With our approach we introduce a framework to exploit the processing power of a platform made of multiple resources of different types including FPGA. By extended StarPU to support FPGA, we inherit a rich task execution environment, were FPGA can be transparently utilized as yet another accelerator.

Chapter 4

An Heterogeneous HPC Framework Integrating FPGA

Contents

The objective of this thesis as introduced in chapter 2 is to deliver:

- A runtime system that provides an adaptable (easy to plug and drive to an existing platform), scalable (multiple FGPA devices per node), portable (not limited to a certain board), non-restrictive (FPGA support does not prevent the concurrent use of other accelerators) and transparent (the developer does not need to be aware of the underlying architecture details) task-level integration of FPGAs in a Distributed Memory architecture, based on a relaxed consistency memory model that provides a global address space to the programmer.

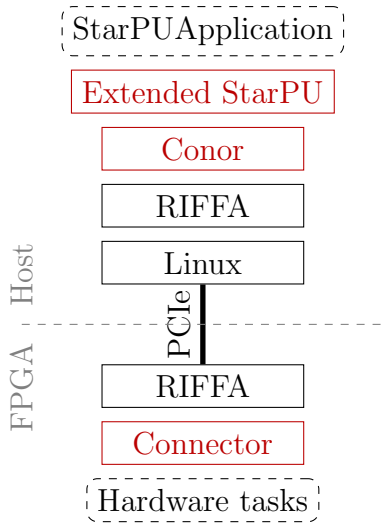
- A case study were an FPGA is used to accelerate parts of an application, showing the impact of the integration to the rest of the parameters taken into account from the scheduling point of view.

- An example of a widely used HPC kernel -*GEMM*, that is optimized to deliver augmented perfromance when executed on an FPGA.

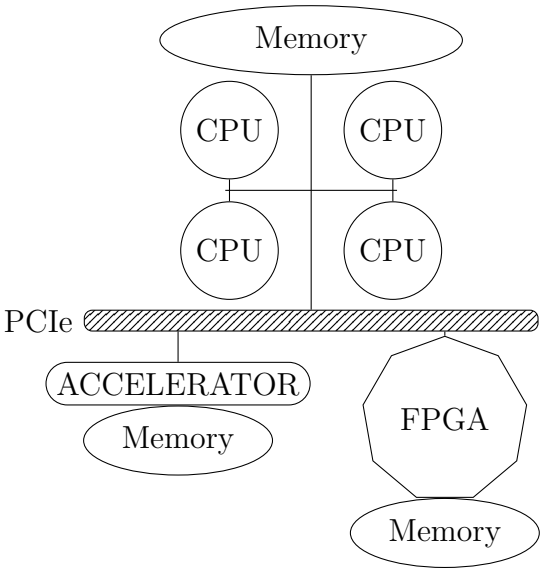
Section 4.1 of this chapter focuses on the first part of the objective, describing the details of the framework that corresponds to the aforementioned properties. Section 4.2 demonstrates the accomplishment of our claim for an easily programmable framework, with a proof-of-concept evaluation of a matrix multiplication application.

4.1 The Proposed Framework

The layout of the framework is presented in figure 4.1a. The figure is vertically split into two parts, *host* and *FPGA*, to illustrate the hardware integration of platform we target and the distribution of the platform components over both the host side and the FPGA side. Solid black boxes correspond to tools and infrastructures that our approach relied on. Red boxes correspond to software layers that either we created from scratch or extended. Finally, dashed boxes correspond to the part that the users will have to develop in order to run their application on a FPGA-based HPC machine.



(a) The proposed framework. RIFFA is responsible accessing of the FPGA through the PCIe bus, using a dedicated RIFFA Linux kernel. On the software side on top of RIFFA we introduce Conor a library responsible for the orchestration of the communication between the host and the device. We extend StarPU to support FPGAs. That way we inherit its runtime level management mechanisms including dependency analysis, scheduling and data exchange automation. On the hardware side we introduce a connecting component between the RIFFA hardware interface and the hardware tasks.



(b) Reference heterogeneous architecture. Our work focuses on heterogeneous nodes composed by a series of CPU cores, accessing the same main memory (assembling the host), and FPGA among other devices used as accelerators. Every device has access to a separate memory address space, hence it is represented with its own memory on the figure.

Figure 4.1: Figures 4.1a and 4.1b present the software stack and the reference architecture of our approach.

Unlike most proposals from the related work, we target FPGA-based platforms in which the host and the FPGA communicate through a PCIe link, since it allows the construction of scalable nodes composed by multiple accelerators. We used RIFFA, described thoroughly in section 4.1.3.1, to enable this communication. In short, RIFFA provides both a software level interface and a hardware handshake protocol, and so appears on both side of the PCIe link on figure 4.1a. For the communication with the hardware tasks on the FPGA side, we built a communication library called Conor on top of RIFFA, that is described in section 4.1.3. Conor provides mechanisms that allow allocating channels, mapping hardware tasks to channels and monitoring the performance of the operated communication.

On the software side, our proposal comes with an extension of the StarPU runtime

system to be able to seamlessly execute data-flow task-based applications on any heterogeneous platform that includes FPGA. Section 4.1.1 describes the use of StarPU as the runtime system for the implementation of the task model and the capabilities that come along with it. In particular, section 4.1.1.1 presents how the task model is expressed and implemented in StarPU ; section 4.1.1.2 analyzes its shared memory model implementation of distributed physical memories ; and section 4.1.1.3 provides an overview of the available task scheduling policies. Section 4.1.2 completes the overview of the software part of the framework, providing details on the *worker* we added to support FPGA.

The hardware side of our framework is presented in section 4.1.4. As mentioned previously, we first provide a detailed presentation of the handshake protocol that comes with RIFFA. Then, we describe the hardware-level protocol that comes with every hardware kernel, and last we describe a hardware component that was designed to provide high performance connectivity between the two interfaces. Section 4.1.4 ends with a complete picture of the final hardware design that corresponds to what will be finally used for the configuration of the hardware fabric.

4.1.1 Inside StarPU

The main focus of this section is to provide the essential information and functionalities that StarPU already delivers, that are necessary to understand the extension we proposed to allow FPGA to be used as processing units. We took the code snippets presented in this section from the source code of StarPU [80]. Some of them have been slightly modified in order to serve the purpose of the listing, which is to give an insight on the functionality of every component we present.

4.1.1.1 The Task Model

We recall our definition of a task in the context of this thesis as a well-defined set of operations imposed on some data. There are three different terms within the terminology of StarPU that are associated to this notion: the *codelet*, the *task* and the *job*. A codelet is an abstraction of a task, implemented as a structure that allows the programmer to define the parameters determined by the abstraction. A task is the instance of a codelet, or more precisely it is the codelet along with the data it will be applied on. A job is a task whose dependencies are satisfied, and is scheduled to be executed to the appropriate processing unit. Listing 5 shows a C-based pseudo-code of a codelet.

```
struct starpu_codelet
{
    uint32_t where;
    int (*can_execute)(unsigned workerid, struct starpu_task
    ↪  *task, unsigned nimpl);
    enum starpu_codelet_type type;
    int max_parallelism;

    // The quotted double per-cent (<%%>) here corresponds
    // to any supported processing unit including
    // cpu, opencl, cuda, mic, scc, conor - for FPGA
    starpu_<%%>_func_t <%%>_func STARPU_DEPRECATED;
    starpu_<%%>_func_t <%%>_funcs[STARPU_MAXIMPLEMENTATIONS];

    char <%%>_funcs_name[STARPU_MAXIMPLEMENTATIONS];

    char *conor_kernel_type[STARPU_MAXIMPLEMENTATIONS];

    int nbuffers;
    enum starpu_data_access_mode modes[STARPU_NMAXBUFS];
    enum starpu_data_access_mode *dyn_modes;

    unsigned specific_nodes;
    int nodes[STARPU_NMAXBUFS];
    int *dyn_nodes;

    struct starpu_perfmodel *model;
    struct starpu_perfmodel *energy_model;

    unsigned long per_worker_stats[STARPU_NMAXWORKERS];

    const char *name;

    int flags;
};
```

Listing 4: C-based pseudo-code of a codelet

The first information held by the codelet is the set of potential processing units that an associated task can execute on. Examples of such processing units are a CPU, an OpenCL or a CUDA device, or a Conor device (a hardware task executed on FPGA.) For every processing unit a task may execute on, the programmer needs to

provide at least one implementation of the body of the task. This is done by providing the function(s) associated to the architecture (lines 11,12 of listing 5). Then what needs to be defined is the number of data structures that an associated task will be applied on, as well as the access modes on every such structure (read-only, write-only, read-write, etc). Lastly, the programmer can determine a performance model associated to the codelet, that will be later described in section 4.1.1.3.

```
1  #define PERTURBATE(a)     ((starpu_drand48()*2.0f*(AMPL) + 1.0f -
   ↪  (AMPL))*(a))
2
3  double cpu_chol_task_gemm_cost(struct starpu_task *task,
4      struct starpu_perfmodel_arch* arch, unsigned nimpl)
5  {
6      uint32_t n;
7
8      n = starpu_matrix_get_nx(task->handles[0]);
9
10     double cost = (((double)(n)*n*n)/50.0f/10.75/8.0760);
11     return PERTURBATE(cost);
12 }
13
14 double cuda_chol_task_gemm_cost(struct starpu_task *task,
15     struct starpu_perfmodel_arch* arch, unsigned nimpl)
16 {
17     uint32_t n;
18
19     n = starpu_matrix_get_nx(task->handles[0]);
20
21     double cost = (((double)(n)*n*n)/50.0f/10.75/76.30666);
22     return PERTURBATE(cost);
23 }
```

Listing 5: Performance model example. The code contains the performance model of the *GEMM* used in Cholesky decomposition form the resources of StarPU.

Abstracting tasks to codelets provides great flexibility to the programmer, since combined with the performance models and the scheduling policies, the identification of the best performing candidate can occur automatically by the runtime.

4.1.1.2 The Memory Model

While StarPU supports multi-node distributed architectures, the focus of our study is put on architectures similar to the one shown in figure 4.1b. In such architectures, multiple CPU cores have access to the same physical memory. The HPC community usually refers to this part of the machine as the *host* part. Multiple accelerators, with their private memories, are connected to the host via a PCIe bus.

StarPU implements a relaxed consistency shared memory model. This means that the runtime provides a global address space, on top of the distributed physical memories to the programmer.

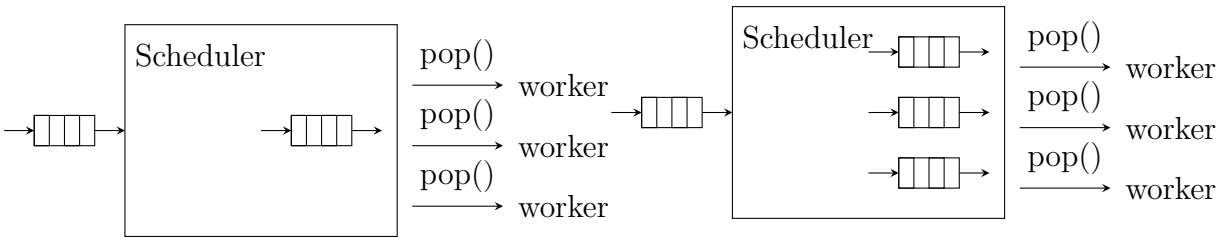
A developer needs to register the data that will later be accessed by the tasks. There are different data types that can be used, from a single variable to 1-D vectors, 2-D matrices and 3-D blocks. StarPU provides a handler to every registered data, which will be used later to associate data chunks with tasks, before the task launching. Consequently, the runtime at every point of the execution is aware of which data are accessed by which tasks with what privileges, and ensures the consistency of the multiple physical memories.

4.1.1.3 Task Scheduling And Performance Models

Task scheduling is in the core of StarPU, and is performed by employing scheduling policies. There is a series of such policies, that come with the runtime system itself, although, the programmer can implement new ones based on the needs of the application to be deployed. As a clarification to avoid misunderstandings, it needs to be declared that the scheduler is the component responsible for the mapping of tasks to the processing units once their dependencies are satisfied. Those tasks are considered to be ready for execution, and are pushed in a ready tasks FIFO (Figure 4.2, left hand queue). Hence the goal of the scheduling policies is to dynamically distribute sets of *ready* tasks to the underlying processing units that can be either *idle* or *busy*. The StarPU execution model relies on the notion of *worker* threads, responsible for executing tasks on specific processing units, like for example a core of a multicore CPU, a GPU device or a Xeon Phi co-processor. Every processing unit of the target platform is represented by a *worker* thread attached to it, running on the host. The dynamic set of tasks to be executed is most of the time implemented as a FIFO queue of tasks, which can be either globally accessed by every worker, as depicted on figure 4.2a, or split into per-worker data structures, like shown on figure 4.2b, depending

on the needs of the scheduling policy. The worker is the one responsible for popping tasks that have been assigned to its associated processing unit by the scheduler.

The available scheduling policies can be separated into two categories. The first one gathers strategies that tries to even the number of tasks to be assigned to each processing unit, like `random`, `eager`, `lws`, `ws`, `prio` and `heteroprio`. The second one involves heuristics that take an estimation of a task *performance*, most of the time expressed as an execution time, on any processing unit to compute a mapping. The `dm` and `heft` scheduling families are examples of performance model-based schedulers.



(a) Centralized queue: In this simple scheduler architecture, there is one main queue that workers access to obtain tasks.

(b) Worker Dedicated Queues: Here every worker has its own dedicated queue with ready tasks, and the scheduler assigns the workload according to the determined policy.

Figure 4.2: Reference architectures of a StarPU scheduler. A scheduling policy is responsible to determine the mapping between ready tasks and the available processing units. The source of the scheduler is a FIFO queue holding the tasks with satisfied dependencies, represented by the left side FIFO on figures 4.2a,4.2b. On the other side there is the amount of the available processing units represented by their workers, performing popping operations on the scheduler so they can proceed to task execution.

The `eager` scheduler comes with a centralized FIFO of ready tasks that workers concurrently pop tasks from. The `random` scheduler randomly distributes ready tasks to per-worker queues. The `ws` and `lws` schedulers implement the *work-stealing* execution model, in which an idle worker tries to steal tasks from other loaded queues in a dynamic fashion. The `prio` and `heteroprio` schedulers use centralized queues of tasks that are sorted according to a per-task user-specified priority.

For the second category where performance models are employed, the majority of the policies require a series of calibration that allows StarPU to obtain reference performance estimations for the execution of a task on a specific architecture. In order to use any of those performance model-based scheduling policies, a cost function needs to be implemented for every architecture and every implementation of a task. Listing 18 of appendix A shows a C-based pseudo-code of the structure of a performance

model, while listing 19 shows an example of a performance model for *GEMM*, used in Cholesky decomposition, for both the CPU and the GPU.

If there is not enough information about the estimated execution time of a task (no cost function associated with the task) on a specific architecture, the scheduler will fall back to an eager scheduling policy.

The performance model-based scheduling policies that are shipped with StarPU are derivatives of HEFT - Heterogeneous Earliest Finished Time [3]. In HEFT, a submitted task can be executed in multiple processing units of potentially different architectures (hence "Heterogeneous"), and the scheduler needs to know which worker will the task be eventually assigned to. The rank of a task n_i for a processing unit u is recursively computed by the formula:

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_j)} (\overline{c_{ij}} + rank_u(n_j))$$

and corresponds to the estimated execution time of the task n_i at the processing unit u. Where $\overline{w_i}$ corresponds to the average estimated execution time of n_i to every processing unit, n_j corresponds to a child of n_i, $\overline{c_{ij}}$ corresponds to the average estimated communication time of the data of n_i that will be consumed by n_j. Once the rank of a task has been computed for every processing unit that can be executed by, it is assigned to the one that ensures the earliest finished time (as the name of the policy denotes).

The most simple HEFT derivative that comes with StarPU is `dm` (*dequeue model*), where tasks are scheduled as soon as they become available to the worker of the processing unit of the earliest estimated execution time.

An evolution of dm, is `dmda` (*data-aware dm*), according to which data exchange is also taken into account. This policy can be considered a deprecated alias of HEFT. Other refinements of the data-aware dequeue model are the `dmdar`, `dmdas`, `dmdasd` schedulers, where supplementary conditions are taken into account, like task priorities or the proximity of reading buffers.

4.1.2 The StarPU FPGA worker

This section reports on how we extended StarPU to support FPGA, with the introduction of FPGA workers, also called Conor workers after the name of the communication library that enables the FPGA accessing, described in section 4.1.3. As explained in section 4.1.1.3, we recall that the worker is the software representative of an underlying processing unit. It is implemented as a POSIX thread and is bound to a CPU

core. The worker is responsible to monitor the availability of the underlying resource, and manage the task execution that is orchestrated by the scheduler.

As already mentioned, StarPU implements a relaxed consistency shared memory model, where data exchange between the main memory and the private memories of the devices is happening transparently from the programmer. In our work, we followed an agile model, with two rounds of development. In the first round, data exchange was controlled by the developer, by a *conor_func* that is written in software. One such function should be provided for every codelet with a FPGA implementation of a task. With this function, the programmer:

- allocates a channel using Conor
- retrieves the data structures associated to the task, and sends them to the device using Conor
- receives and distributes the output to the appropriate place of the host memory using Conor
- allocates the channel using Conor

For the exchange of input and output, there is a certain order that needs to be respected; data dependencies should be sent from the host to the device in the order depicted in the design of the hardware implementation of the task. The experiments of sections 4.2 and 5.1 have been conducted using this version of the framework.

In the second round off development, we proceeded into supporting automatic data exchange. The main memory model of StarPU is described in section 4.1.1.2, where it is stated that StarPU uses a MESI-like protocol to ensure consistency between the different physical private memories. The integration of FPGA to this model is not straightforward because of the following reasons:

- Conor can only transmit memory blocks that are contiguously stored in the host memory
- Data up-streaming should be completed using a single call to Conor
- Exchange data sent to the device are private to the task and can not be re-used.

For contiguous blocks, down-streaming is a trivial process, since a simple call to Conor, using as a source buffer the one that the data are originally stored, would do the work. Nevertheless, this operation does not come by default at a zero performance overhead, since RIFFA driver locks the associated memory pages during

the transaction. For non-contiguous data blocks though, the performance overhead of initiating a new transaction for every contiguous data chunk would kill the overall performance. For that reason, in the second phase of integration, we allowed Conor to allocate memory buffers, associated to every data structure a task is associated with, through which we perform the actual data exchange. Those Conor buffers exist on the software side, and are allocated once for every data matrix, block, vector or variable.

According to RIFFA, on every up-streaming operation, the number of words sent from the FPGA side should match the number of words received from the host side. In case the buffer size of the software end is smaller than the amount of transmitted words, the remaining data will be discarded. This limitation complexes a lot an automatic receiving operation, in cases where a task has several output dependencies.

Lastly, there are several limitation by the fact that the data on hardware tasks are stored in BRAM blocks. As mentioned previously, those blocks are private to every task, and can not be re-used between rounds of execution. In other words, the hardware side is not aware of which data exists on witch block, and on every task execution, the IO phase needs to be repeated. This means, that regardless of weather the Conor buffer is holding a valid image of the data, on every round of execution, task dependencies need to be re-sent to the hardware task.

To accomplish automatic data exchange, StarPU worker uses the Conormemory allocation mechanism, the same way as an OpenCL buffer is allocated by the OpenCL worker on the device. The main difference here is that the allocated memory exists on the host and not on the device side. Once a task is assigned to the worker by the scheduler, the worker will first ensure that there is an implementation of this task for its architecture. Then it schedule the execution of the job associated to the task:

- it will allocate a channel corresponding to the assigned task using Conor
- it will communicate with the StarPU datawizard (the component of StarPU responsible for the memory consistency) and allocate the Conor buffers when necessary (on the first data access)
- it will send every input dependency from the Conor buffer to the hardware side using Conor, respecting the order by which dependencies are declared to the codelet (first declared first sent)

- it will create a temporary buffer, fetch the output dependencies of the task and distribute them to the appropriate Conor buffers, again respecting the declaration order.
- it will release the channel.

The consistency of the Conor buffers is managed by the StarPU datawizard. The assignment of tasks to the worker is managed by the StarPU scheduler, respecting the selected scheduling policy.

In our approach, the number of Conor workers that need to be created corresponds to the number of hardware tasks synthesized on the hardware design used for the configuration of the device. This is due to the fact that every hardware task is completely independent, and can be concurrently accessed by a software thread in order to provide the associated data enabling the execution of its core.

Listing 17 of appendix A shows the main job of a worker, in the format of a C-based pseudo-code

4.1.3 Conor: A Communication Library Based On RIFFA

Conor is a communication library, that was created within the scope of this thesis, aiming to provide a complete infrastructure to access and monitor the behavior of all the FPGA devices connected to the machine. Device accessing corresponds to a set of utilities including data exchange mechanisms and communication performance report.

Conor relies on RIFFA [65], which stands for *Reusable Integration Framework for FPGA*, that implements a channel-oriented approach where software threads communicate with hardware tasks through individual channels over PCIe. Each channel comes with its own lock a thread must acquire to guarantee exclusive access to the corresponding hardware tasks. The responsibility to acquire and release the lock for accessing a particular channel remains to the user though.

4.1.3.1 RIFFA

RIFFA as a PCIe accessing framework satisfies the requirements for scalability and portability that were set in our objective. It provides support for up to five FPGA devices connected via PCIe to the host node. RIFFA supports a series of boards from both Xilinx and Altera. The version of RIFFA used in our framework is 2.2.2, and provides access to FPGA connected over PCIe generations gen1 or gen2. From the

performance point of view, it was designed ensuring a minimal interaction between the host memory and the PCIe bus, in order to obtain minimal communication latency. Regarding the communication throughput, performance results indicate that burst data exchange using RIFFA can reach to the point of bus saturation for both gen1 and gen2.

Architecturally, RIFFA provides a software library that is linked with the host side of the application as well as a hardware design where the developer can integrate the hardware tasks corresponding to the application. Both of them will be thoroughly described later in this section. On the host side, the software library cooperates with a kernel driver that is loaded from the operating system during startup. The driver holds a series of 4MB of PCIe-accessible buffers, in kernel space, that are used in order to transfer data from and to the device. Data exchange happens using DMA, allowing the CPU core that invoked the exchange to be utilized by another software thread.

When the software library invokes *down-streaming*, which refers to the action of sending data from the host to the device, a driver buffer is allocated. The corresponding channel is notified for the exchange, data are copied from the user memory to the allocated buffer, and then transferred through the channel to the hardware tasks. Then, the software thread is notified that the data exchange has been completed and the driver buffer is freed. RIFFA only supports blocking communication, meaning that the software thread blocks until it receives an acknowledgment that the communication has been completed. When the software library invokes *up-streaming*, which refers to the action of sending data from the device to host, the hardware task sends a signal to the channel that a data sending is ready to occur. The channel then notifies the driver, that allocates an internal buffer for the data copy, and returns the address to the channel. Once data are written to the internal buffer, the hardware task is notified, and data are copied from the internal buffer to the user space. The host thread receives an acknowledgment that data writing has been completed, and the internal buffer is freed, while the thread can resume its execution.

The next two paragraphs provide a thorough description of the software and hardware interface of RIFFA to the programmer.

Software Side Interface During application development, for the host code, it is safe to assume that the device is configured with a hardware design of the expected behavior. What the software side of RIFFA provides is a set of functions that enable to open the device, send and receive data to and from a channel and close the device.

Invoking *fpga_list()* in the beginning of the execution populates a dedicated data structure that contains the number of configured FPGA devices, as well as the number of active channels per device. Before any data exchange, the user needs to invoke *fpga_open()* on every FPGA device detected by *fpga_list()* which returns a device handler, that will later be passed to *fpga_send()* / *fpga_recv()* in order for data to be sent to / received from the proper destination device.

Regardless of the direction of the data, on the host, data will be held on a buffer of the software thread invoking the call to the kernel driver. In the case of data sending, data should be contiguously stored in a *void** buffer, while in the case of data receiving they will be contiguously stored in a *void** buffer respectively. For both data sending and receiving the user needs to determine the expected payload size of the exchange, which corresponds to the number of 4-byte words that will be read or written to and from that *void** buffer. In the receiving scenario, if the size of the buffer is not sufficient to hold the entire payload, the remaining data will be discarded.

Hardware Side Interface On the hardware side, there is a handshaking interface for both the up-streaming and the down-streaming, that IPs can use in order to access the PCIe bus. On the back-end of this interface there is a receiver (RX) engine for the down-streaming path, and a transmitter one (TX) for the up-streaming path. Both are multiplexed to provide up to 12 bidirectional channels at the endpoint. Every channel may operate at different frequencies, determined by the IP that is connected with. RIFFA provides template designs for user clocks between 5 and 250Mhz.

On the hardware part, data are encapsulated into packages. The size of a package is associated to the width of the PCIe bus, that is set as a parameter in the configuration of the hardware part of RIFFA. In the current version of our framework, we used PCIe bus sizes of 64 or 128 bits.

In the case of down-streaming, the channel raises a `CHNL_RX` flag indicating that an exchange is to be performed. The IP needs to acknowledge the notification raising a `CHNL_RX_ACK` signal. Information about the upcoming exchange is provided by the RX engine regarding the length of the transaction, or a possible offset of the data, as well as whether this is the last of a series of transactions. RIFFA indicates that `CHNL_RX_DATA` contains valid data with the `CHNL_RX_DATA_VALID` flag, that can be consumed by the IP with a read-enable flag `CHNL_RX_DATA_REN`, at a pace of one package per clock cycle.

An equivalent handshaking procedure takes place on the up-streaming exchange. It is the IP who is responsible to indicate the initialization of a transaction by raising `CHNL_TX`, a flag that needs to be maintained high until the exchange completion. Once the TX engine acknowledge the up-streaming, the IP needs to provide aforementioned information (`CHNL_TX_LEN`, `CHNL_TX_OFF`, `CHNL_TX_LAST` and indicate the beginning of the actual transaction by raising `CHNL_TX_DATA_VALID`. The engine can then consume one package per cycle raising a `CHNL_TX_DATA_REN` flag.

4.1.3.2 Channel Allocation

As mentioned in 4.1.3.1, RIFFA splits the PCIe bus into individual bidirectional channels, that can be accessed concurrently to communicate with the hardware tasks. However, if one individual channel is accessed by several software threads at the same time, then the behavior of the driver is undefined. Moreover, considering that in our architecture there is a one-to-one mapping between the active channels and the hardware tasks, we need to ensure that it is the same worker thread on the software side that is communicating with a given hardware task. To address potential hazards that derive from the aforementioned concepts, we implemented in Conor a mechanism to allocate channels on the software side.

Once a software binary is linked with Conor, *fpga_list* is called in order to obtain the number of FPGA devices connected via PCIe on the platform, as well as the number of active channels for every device. For every channel of every device, a lock of type *pthread_mutex_t* is created and initialized. Acquiring this lock will provide exclusive access to the corresponding channel, and by extent to the corresponding hardware task.

Conor users have two ways to allocate channels, illustrated by two different scenarios. In the first scenario, multiple instances of the same hardware task are configured to every connected FPGA device. In this case, every channel is equivalent from the programmer point of view. In other words, in order to execute a hardware task, we need to allocate any of the available/idle RIFFA channels. For this purpose, Conor provides the *conor_allocate_channel* function, which, when called, cycles through all the available channel locks (with *phtread_mutex_trylock*), until a free channel lock is found and obtained. Once the lock is acquired, the corresponding channel is returned to the calling thread that can further continue its execution. Every allocated channel needs to be released once the execution of the hardware task is finished. A task is considered to be finished once the data that it is producing have been read by the software thread. It is the responsibility of the programmer to instruct Conor to release

a channel on the software side using the *conor_release_channel()* function provided by the library.

In the second scenario, different types of hardware tasks are configured to the FPGA devices. The procedure of channel allocation for this scenario will be described in the next section.

4.1.3.3 IP Mapping

An FPGA can be configured with an arbitrary number and type of hardware tasks within the limits of the twelve available channels provided by RIFFA. In our approach we only consider static configurations, where the hardware design is provided statically by the user, and flashed to the device before the beginning of the execution. During the execution, a software thread must allocate a channel of the type of the task it wants to execute. To address this, Conor provides the ability to allocate a channel of a certain type, calling the *conor_reserve_channel_of_kernel_type()* function.

In order this to be possible, the user needs to provide a description of the configuration of each device, using an environment variable. The user indicates the kernel types in channel ascending order, using a colon separation format, like the one indicated in Listing 6. In this example, there is only one FPGA device connected to the system, with four active channels, where channels *0 and 1* correspond to kernel type *1* while channels 2 and 3 to kernel type *2*.

```
1 FPGA_CONFIGURATION="kernel1:kernel1:kernel2:kernel2"
```

Listing 6: Example Configuration

In the scanning phase at the beginning of the execution, Conor verifies that the amount of active channels is compatible with the description of the configuration provided by the user. Once the checking is successfully completed, Conor groups channels by their kernel type. In the example of code on listing 6, two groups will be created, one containing channels 0 and 1 associated to kernel type *1*, and another containing channels 2 and 3 associated to kernel type *2*.

In a scenario where a task of type 1 needs to be executed, a software thread needs to allocate a channel associated to `kernel1`. Once the function *conor_reserve_channel_of_kernel_type()* is called, Conor cycles through the channels of the corresponding group (group 1), and acquires the lock of the first available/idle channel of the group. In alignment with the basic channel allocation process of section 4.1.3.2 once the

execution of the hardware task is finished, the user needs to release the channel calling *conor_release_channel()*.

4.1.3.4 Performance Reports

As described in section 4.1.3.1, once *fpga_send()* or *fpga_recv()* is invoked, RIFFA returns the number of 4-byte words that have been successfully transmitted over the channel specified by the call. If there is no error during the design and the execution of the application, the number of exchanged data should be equal to the number indicated by the user within the function call.

From the programmers point of view, it is convenient to have easy access to the information regarding the behavior and the performance of the communication with the device for profiling, monitoring and performance debugging purposes. For example, in the case of StarPU integration, this information could be used by a performance model-based scheduler. In order to provide such utility, for every data exchange invoked using Conor, a tuple is returned to the calling thread, holding the actual size of the exchanged data as well as the time of the data transaction. The transaction time refers to the time the thread spent idle, waiting for the completion of the transaction. This time may or may not correspond exclusively to data exchange, and this derives directly from the reference model of a hardware task from the software point of view.

According to the aforementioned hardware task model, the lifespan of a task is composed by three phases: a first one dedicated to receive the input data, a second one in charge of the computation itself and a last one responsible for sending back the output data.

In the case of *conor_data_send()*, the measured time corresponds strictly to the time that the thread remained idle because of the undergoing data exchange. In the case of *conor_data_recv()*, there is no guarantee that the computation phase of the application has been completed before the invocation of the exchange. Since data sending and receiving are blocking, the time returned by Conor corresponds to the time during which the thread remained idle waiting for the completion of the transaction, which from the hardware point of view could be spent both in the task execution and the data exchange.

4.1.4 Hardware Level Integration

Sections 4.1.1 to 4.1.3 were dedicated to the presentation of the development of the software side, along with the hardware end-point of RIFFA for purposes of complete-

ness. This section is dedicated to the hardware part of the framework, where the analysis is focused on the characteristics of the hardware components assembling the design. Section 4.1.4.1 presents the design requirements regarding the input and output interfaces of a hardware task in order to be compatible with the rest of the design. Section 4.1.4.2 exposes the internals and the behavior of the connector, the component between a hardware task and a RIFFA channel hardware end-point. Lastly, section 4.1.4.3 provides a complete overview of the hardware design, where the hardware side of RIFFA, the hardware tasks and the connectors are all assembled together.

4.1.4.1 IO Protocol For The IPs

The focus of this work is to provide a programmable framework for the high performance computing community. It is obvious that developers with this expertise will not attempt to design their hardware kernels at the RTL level, but will most probably use one of the available High Level Synthesis tools introduced in section 3.2. In this case, they are bound to the input and output interfaces provided by the chosen High Level Synthesis tool.

Most of them implements a simple and widely-used directionless two-way handshake protocol. In a scenario where one hardware IP is acting as a sender, and wants to communicate with another one acting as a receiver, two control signals are needed in total. Besides the control signals, the actual *DATA* bus needs to be instantiated. This bus will be used to pass data from the sender to the receiver. The sender raises a *READY* flag once the *DATA* bus contains valid values, which is considered by the receiver as an indicator of a data exchange transaction. The receiver then raises an acknowledgment *ACK* flag, and data are consumed word after word, at a pace of one word per cycle. The size of a word is determined by the width of the *DATA* bus.

Vivado HLS provides such handshake interface, named *ap_hs* (Access Point Hand Shake), which we eventually use. In case that buffering can better serve the hardware design, Vivado HLS provides a number of FIFO implementations, that can be incorporated to the rest of our framework with minimal changes.

4.1.4.2 Description Of The Connector

What we call *connector* here refers to a hardware component meant to link a hardware task to a RIFFA channel hardware end-point. It was created due to the difference of the communication protocols between the implementation of the hardware end-point of RIFFA and IPs created by a High Level Synthesis tool. Designing this component implied to take architectural decisions to make it easy to integrate to the existing

framework, while imposing the smallest overhead possible on the performance of data transaction, both in terms of latency and throughput. The integration part concerns the distribution of the available RIFFA channels to the hardware tasks. We already mentioned in section 4.1.3.1 that RIFFA allows the PCIe bus of every FPGA device to be split into up to twelve, completely autonomous and concurrent bi-directional channels. We recall that, in our framework, we have chosen to associate every hardware task with one channel. There is a trade-off over this decision. Indeed, associating more than one channel to a single hardware task would enable faster data sending, since input data would be able to be sent concurrently. However, this would impose a stronger limitation to the number of hardware tasks that can be synthesized on a single device, from one task per channel to one task per group of channels. Moreover, associating more channels per task would require several software worker threads associated to a single hardware task. Such a design would greatly increase the complexity of the design of the Conor driver of StarPU, while the improvement of the overall performance is not guaranteed, since the overhead on the software side can not be computed easily.

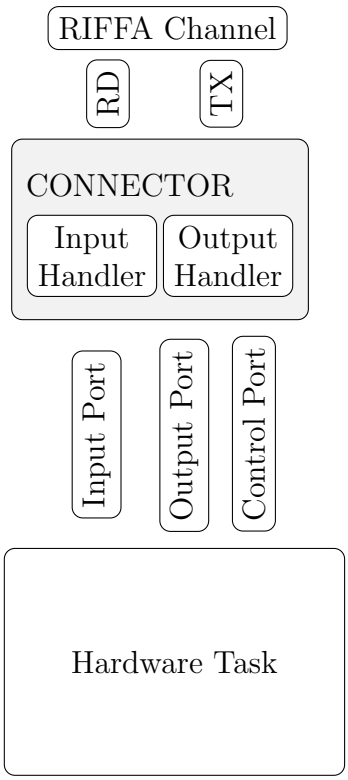


Figure 4.3: Connector: a RIFFA channel to hardware task adapter. The connector is assembled by two individual components the Input and the Output handler. The input handler is responsible to drive the RD direction of the RIFFA channel to the input port of the hardware task. The output handler is responsible to drive the output port of the hardware task to the TX direction of the RIFFA channel, using information provided by the control port of the task.

Figure 4.3 shows how the connector interfaces an hardware task with a RIFFA channel. The input and the output paths are implemented with two different handlers. Every handler is a finite state machine that performs the control translation between the two protocols. The input handler will raise the `CHNL_RX_ACK` signal when `CHNL_RX` is indicated by the `RX` part of the channel, and raise the `READY` flag at the end-point of the hardware task. It will then raise the `CHNL_RX_REN` flag when the `ACK` flag of the hardware task is on.

The output handler is a little more complicated, since the `TX` end-point of the RIFFA channel requires the length, in number of words, of the upcoming transaction, which is not held by the *ap_hs* interface. This information is provided by the task, through the control port, which is implemented in the form of another *ap_hs* interface. It is the programmer responsibility to propagate the length of data up-streaming through the control port prior to the initiation of the transaction itself.

4.1.4.3 Overview Of The Entire Design

This section provides a complete overview of the reference hardware design of our framework. While one could infer the structure from the preceding sections, the goal of this section is to clarify any potential blurry point.

Figure 4.4 shows the design of the FPGA fabric, along with its accessing point, which is the PCIe bus. Direct access to the bus, is provided via designs provided by the FPGA vendors, like the PCIe interface provided by Xilinx that we used for the series of our experiments. The RIFFA hardware endpoint is connected on top of the vendors-provided accessing component, constituting the entry point for the programmers hardware design. Up to that point, everything is static and predetermined, as nothing here depends on the application our framework will execute. The only thing that needs to be configured by the developer is the width of the PCIe bus, which can vary from 64 to 128bits.

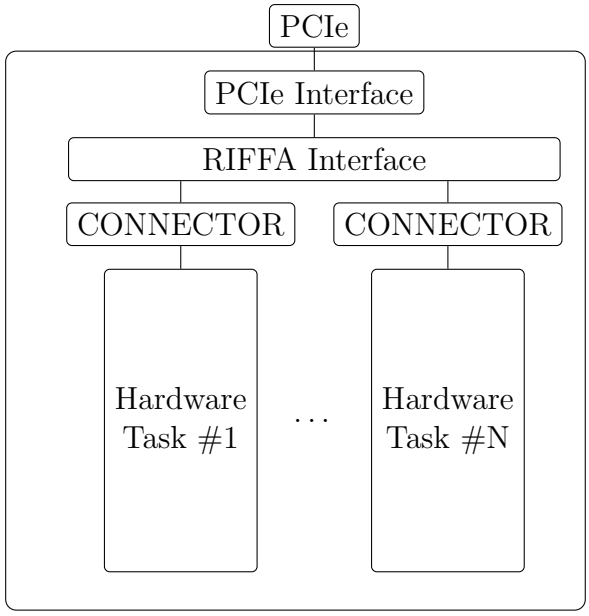


Figure 4.4: Hardware Design Overview: This is a complete layout of a FPGA configuration compatible with our framework. The objective is to provide accessibility to a number of hardware tasks from the software (host) side. The link between the host and the device is the PCIe bus, on top of which the hardware interface of RIFFA is synthesized (using the PCIe interface from the FPGA vendors). Every synthesized hardware task is coupled with a connector component which is associated to a RIFFA channel.

The truly reconfigurable part of our design comes below the RIFFA channels. Indeed, the developer needs to :

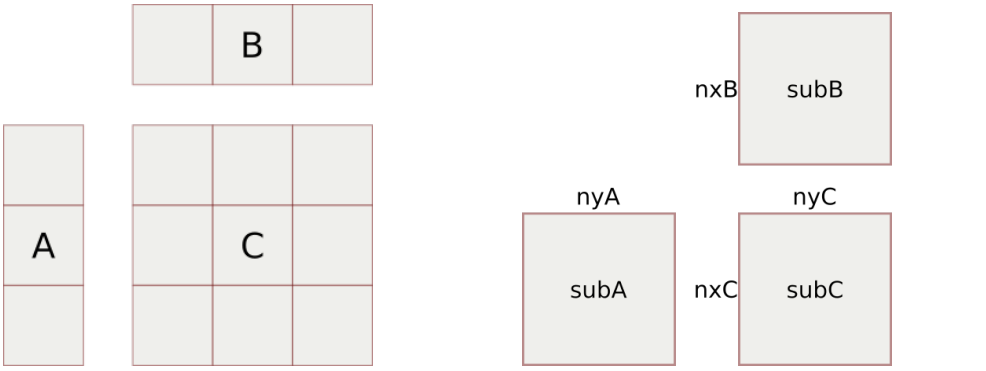
- implement and instantiate the hardware tasks that are going to be flushed to the device ;
- instantiate a connector between every task and the corresponding channel.

Along with the number and the type of hardware tasks to be synthesized, the developer also needs to determine the clock frequency associated to every task. Available clock frequencies vary from 10 kHz to 250 MHz.

4.2 Evaluation

This section presents the first evaluation of our framework on a blocked version of matrix multiplication using an FPGA as an accelerator. This section layout is two-fold. First, section 4.2.2 focuses on the impact our framework has on programmability of HPC platforms including FPGA illustrating the transparency of integration of FPGA compared to the rest of the supported accelerators. Then, section 4.2.3 focuses on evaluating the performance overhead that comes with the extension of StarPU, presented in section 4.1.2.

4.2.1 Blocked Matrix Multiplication



(a) Matrices are partitioned on the horizontal and the vertical axes, creating the blocks. We focused our experiments of orthogonal set-ups; every matrix dimension is identical, and matrices are partitioned homogeneously on every dimension.

(b) In our blocked version of matrix multiplication, a task is associated to the computation of one block of the output matrix C. The input dependencies of the computation are the associated blocks of matrices A and B.

Figure 4.5: Overview of our evaluation algorithm: Blocked version of matrix multiplication. Figure 4.5a shows the partitioning of the input and output matrices while figure 4.5b shows the data blocks associated to a single task.

Figure 4.5a shows the data layout corresponding to the algorithm, whose mathematical form is:

$$C = A * B$$

where A is a $x \times z$ matrix, B is a $z \times y$ matrix and C a $x \times y$ matrix.Dimensions x and y are partitioned in nx and ny number of slices respectively. The workload of a task is associated to the computation of a single block of matrix C. Figure 4.5b shows the blocks associated to a single task. The input dependencies of a task are the associated

blocks of matrices A and B. The output dependency of a task is the C block. In the case of the blocked matrix multiplication we use for the framework evaluation in this section, tasks are completely independent, since there is no task reading from the output of another one. Nevertheless, the framework is designed to support more complicated applications with task dependencies, since they are natively suported by StarPU.

4.2.2 The Programmers' Side

To describe this application using our framework, the application developer must write both the StarPU application along with the hardware implementation of its tasks as shown by dashed boxes in figure 4.1a.

4.2.2.1 The StarPU Application

First, the programmer must describe the codelet for the tasks computing one block in the result matrix. Listing 7 shows how to do that using StarPU's C API. This codelet provides both a CPU implementation and an FPGA one. It also specifies that the task has two inputs and one output through the `nbuffers` and `modes` attributes.

```
1  static struct starpu_codelet cl = {
2      /*
3          cpu_mult (resp. fpga_mult) points to a C function
4          implementing the CPU (resp. FPGA) version of the task.
5      */
6      .cpu_funcs =  {cpu_mult},
7      .conor_funcs = {fpga_mult},
8      .nbuffers = 3,
9      .modes = {READ, READ, WRITE}
10 };
```

Listing 7: Task codelet for a task computing one block of the result matrix. The task has two inputs and one output. The codelet provides both a CPU implementation and an FPGA one.

Listing 8 shows the implementation of the function *fpga_mult*, declared in the definition of the codelet, as the one associated to the execution of the Conor device implementation of the task. A Conor device is a hardware task, configured in an FPGA and accessed via Conor. This function is mainly divided in four steps:

- Instruct a RIFFA channel reservation associated to a hardware task through Conor;
- Retrieve the blocks associated to that particular instance of the task;
- Ask Conor to send the input blocks to the hardware task through the reserved channel;
- Retrieve and copy back into the C block the result of the task computation through Conor;
- Release the reserved channel, enabling another instance of a task of this kind to be executed in this resource.

In this function, the order used to send the inputs is crucial. It must be aligned with the expected inputs order in the hardware task implementations as explained in Section 4.2.2.2.

```
void fpga_mult(void *data[]) {
    /* Ask Conor for a channel, or equivalently for a hardware task */
    int chnl = conor_reserve_a_chanel();

    /* Get data from STARPU */
    int* subA = SPU_MATRIX_GET_PTR(data[0]);
    int* subB = SPU_MATRIX_GET_PTR(data[1]);
    int* subC = SPU_MATRIX_GET_PTR(data[2]);

    /* Get info on which part of the data the task must operate */
    uint32_t nyA = SPU_MATRIX_GET_NY(data[0]);
    uint32_t ldA = SPU_MATRIX_GET_LD(data[0]);
    // Same for B and C

    /* Send A and B */
    int buf_s[nyA], buf_r[nxC*nyC];
    conor_trans sent, recv;
    for (uint32_t j = 0; j < nxC; j++){
        for (uint32_t k = 0; k < nyA; k++)
            buf_s[k] = subA[j+k*ldA];
        conor_data_send(chnl, buf_s, nyA);
    }
    for (uint32_t i = 0; i < nyC; i++){
        for (uint32_t k = 0; k < nyA; k++)
            buf_s[k] = subB[k+i*ldB];
        conor_data_send(chnl, buf_s, nyA);
    }

    /* Receive C. This is blocking */
    conor_data_recv(chnl, buf_r, nxC*nyC);
    for (uint32_t i = 0; i < nxC; i++){
        for (uint32_t j = 0; j < nyC; j++)
            subC[j + i*ldC] = buf_r[i*nyC+j];
    }
    conor_release_chanel(chnl);
}
```

Listing 8: Definition of the fpga_mult function.

In that version of our framework, data exchange was not yet automatic. In the current version, data exchange is happening transparently for most of the data formats including a single variable, a vector, a block or a matrix. Said differently, the runtime, and more precisely Conor worker, will automatically send and receive the inputs and outputs of a task, resolving its dependencies once they are fetched by the scheduler.

Then the programmer must write the main part of the application responsible for allocating matrices, register them with StarPU, launch tasks and wait for their completion as shown in listing 9.

```
/* Init and regist A, B and C */
init_and_register_data();
/* Partition data into blocks */
partition_data();
/* Submit all tasks */
ret = launch tasks();
/* Wait for termination */
starpu_task_wait_for_all();
```

Listing 9: Main part of the StarPU application.

The procedure for launching the task is completely uniform; it does not know anything about task types as shown in listing 10. It consists of creating the tasks, associating them with the codelet defined in listing 7, setting their dependencies, and finally submitting them.

```
for (uint32_t x = 0; x < 9; x++) {
  for (uint32_t y = 0; y < 9; y++) {
    spu_task* task = spu_task_create();
    task->cl = &cl;
    /* Get handlers for each block */
    task->handles[0] = spu_get_sub_data(
      A_handle, 1, y);
    task->handles[1] = spu_get_sub_data(
      B_handle, 1, x);
    task->handles[2] = spu_get_sub_data(
      C_handle, 2, x, y);
    starpu_task_submit(task);
  }
```

Listing 10: Submit tasks to the StarPU runtime.

4.2.2.2 Writing Hardware Tasks

On the FPGA side, the application developer must provide the implementation of hardware tasks. The HEAVEN framework relies on Vivado HLS for this task. Using this tool, the application programmer describes the implementation of its hardware tasks in C++. This description consists in:

- Receive the block from A and B;

- Compute the block of C as the product of the blocks from A and B;
- Send this C computed block.

For receiving inputs and sending outputs, the application developer is provided two high-level objects which type is Vivado HLS *hls::stream*. The interface is of type *hls::stream*, and the protocol implementing the streaming interface is the *ap_hs*, that was mentioned in section 4.1.4.1. The first one is used to receive inputs and the second one to send outputs. The function also has a third *hls::stream* parameter used for control. The interface between these three high-level objects with the PCIe bus is handled transparently by the framework through the connector and RIFFA as described in section 4.1.4.

For the receiving part of the function, the programmer must read the blocks for the A and B input matrices in a given order and save them in local variables which are automatically translated to BRAM by Vivado HLS. This reading order must be respected in StarPU when inputs are sent to the FPGA.

For the computation step, the application developer has first to declare the C matrix as a local variable that will also be allocated into BRAM by Vivado HLS. Then the effective computation of C is done using three nested for loops just as in a software implementation.

Finally, the C matrix must be sent back to the host side using the second *hls::stream* object provided to the function. In cases of tasks with several outputs, as for the inputs, the sending order must be respected on both sides.

4.2.2.3 Generating The Bitstream

Once the RTL description of the hardware tasks has been generated with Vivado HLS, the application developer must generate the final bitstream to be used to configure the FPGA. For that task, the HEAVEN framework provides a parametric wrapper written in Verilog. This wrapper is only configured by the application developer to specify the number of hardware tasks to be instantiated. The wrapper combines the specified number of hardware tasks along with the hardware part of RIFFA, and the connector between RIFFA and the hardware tasks. The application developer then uses Vivado to synthesize the complete design to the final bitstream.

4.2.3 Results

We executed the experiments in a heterogeneous machine, where an FPGA was connected to the host via PCIe. The host CPU was a 64-bit Intel Xeon W3530, a

two-way hyperthreaded quad-core, for a total of 8 virtual cores, running at the speed of 2.80GHz, with a smart cache of 8MB, 256KB of L2 and 8192KB of L3, a thermal design reference power of 130W, and a semiconductor size of 45nm. It was coupled with 12GB of DRAM at 1333MHz.

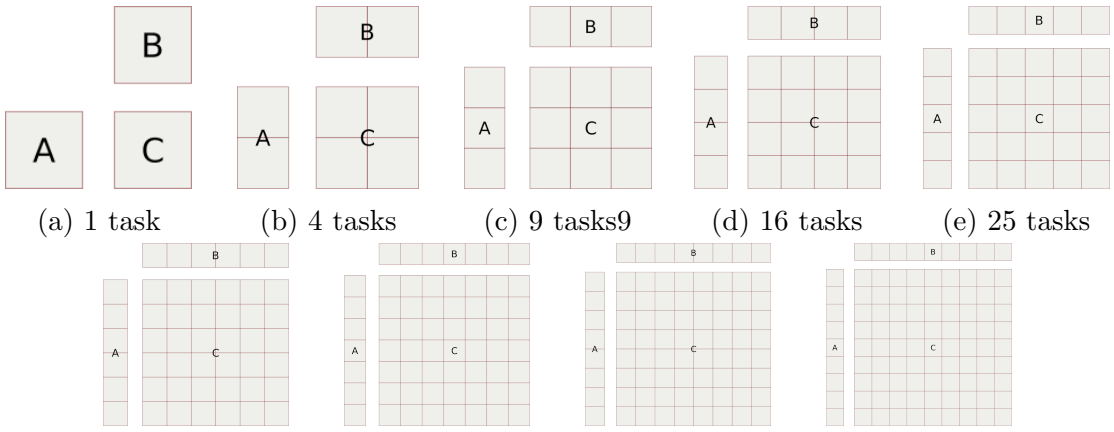
The FPGA we used for our experiments was Xilinx Virtex-7 VC709, a board using the XC7VX690T chip, with 693,120 logic cells, 3,600 DSP slices, 52,920kb of BRAM memory, up to 4GB of RAM at 1866Mbps, and an 8-lane PCIe edge connector.

Within the context of this study, we opted to evaluate the performance we could obtain using our platform on the blocked version of matrix multiplication introduced in section 4.2.1. For the given analysis, we did not consider heterogeneous scenarios, where the scheduler could use within the same execution CPU and FPGA implementations for the tasks. Thus, for every experiment under test, we went through two executions, one using only FPGA tasks and one using only CPU tasks. In this range of experiments, we did not include scenarios where tasks are executed concurrently in both of the resources (the CPU and the FPGA), because our focus was to demonstrate the correctness of our integration.

We expected that the communication overhead would significantly downgrade the performance of the FPGA version. As a consequence, to study this effect, we focused on two constant task sizes for the entire range of experiments. A *low-weight* task operates on blocks of 64x64 integers while a *heavy-weight* task operates on 256x256 blocks.

We also vary the size of the input matrices A and B. We evaluated nine different sizes as shown in figure 4.6. For each one of these nine scenarios, the number of tasks that can be deduced from the way the input matrices were sliced. Each scenario is further divided in two by using either low-weight of heavy-weight tasks. For example, the scenario in figure 4.6e corresponds to a setup for the 64x64 task size and to another setup for the 256x256 one. In this scenario, for both 64x64 and 256x256 setups the A and B matrices are split into five slices leading to a total of 25 tasks. In the 64x64 setup the size of A is then 320x64 and the size of B is 64x320. In the 256x256 setup the size of A is then 1280x256 and the size of B is 256x1280.

In all our experiments, the FPGA was configured with four instances of the hardware task computing a single block of the result matrix. Hence for both architectures, CPU and FPGA we had an equal amount of independent computational entities because in the CPU case, StarPU allocates by default a number of workers equals to the number of physical cores in the host (4 in our case).



(a) 1 task (b) 4 tasks (c) 9 tasks9 (d) 16 tasks (e) 25 tasks

(f) 36 tasks (g) 49 tasks (h) 64 tasks (i) 81 tasks

Figure 4.6: We conducted a series of experiments with a varying matrix size. We kept the task granularity constant across the experiments, modifying the number of tasks launched on every execution by adjusting the size of a matrix dimension and the number of partitions per dimension. For every set-up (configuration displayed in figures 4.6a through 4.6i) we run a case for the heavy weight task and another for the light weight one.

	Low-weight Task	heavy-weight Task
FPGA	1.40 msec	7.01 msec
CPU	1.63 msec	113.37 msec

Table 4.1: Performance of a single task for each architecture and task size.

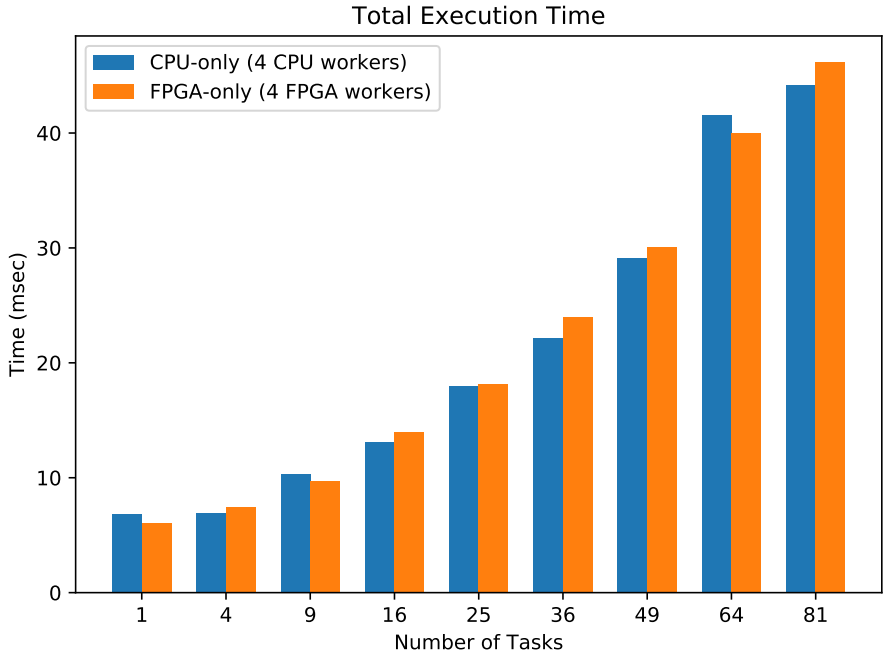
We first evaluated the per task execution time for each architecture. This corresponds to the scenario presented in figure 4.6a. In that case, the computation of the block corresponded to the entire application. The per-task performance we obtained is presented in 4.1. The bigger task shows a significant performance difference when executed on an FPGA compared to when executed on a CPU. In the smaller task, the ratio between the computation and the communication is small, resulting in comparable behavior for both architectures. With the increase of the task size, the cost of data transfer (linear) has a minor impact compared to the cost of the computations (cubic).

Then we evaluated how the extended StarPU runtime behaves when the number of tasks increases. In this case, the expected total execution time should theoretically, follow the formula:

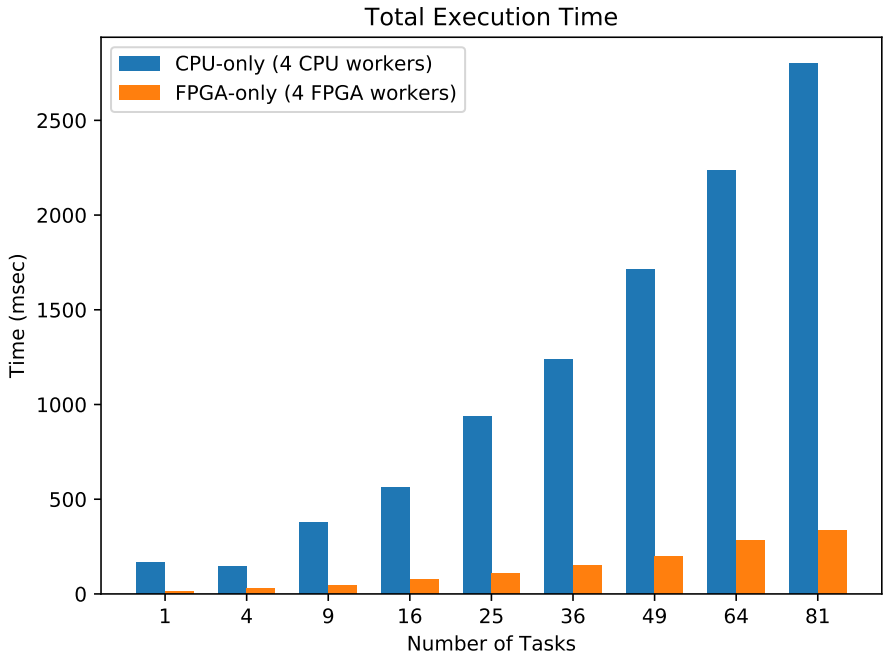
$$t_{total} = \left\lceil \frac{\#tasks}{\#conTasks} \right\rceil \times t_{pt} + \alpha \tag{4.1}$$

In this formula, t_{total} refers to the total execution time of the application, t_{pt} to the execution time of a single task which value is shown in Table 4.1, $\#tasks$ to the

number of tasks, $\#conTasks$ to the number of concurrent tasks and α to a constant representing the initialization overhead. Within our experiments, $\#conTasks$ equals 4.



(a) Results on the low-weight scenario — Orthogonal 64 × 64 Blocks



(b) Results on the heavy-weight scenario — Orthogonal 256 × 256 Blocks

Figure 4.7: Performance results on homogeneous execution scenarios of blocked matrix multiplication for block sizes of 64x64 (figure 4.7a) and 256x256 (figure 4.7b). An execution can happen completely on the CPU cores (CPU-only bars) or the FPGA (FPGA-only bars). In the case of small granularity, the execution time for both architectures is similar while in the heavy-weight tasks, the FPGA impelmentation outperforms the CPU one. In both cases, the increase in the number of tasks shows no performance overhead imposed to the execution time.

Figure 4.7 shows how the total execution time evolves with the number of tasks both for the CPU only version and the FPGA only version. For both low-weight and heavy-weight tasks, Figures 4.7a and figure 4.7b, we observe a linear evolution of the total execution time validating the theoretically expected one. We can then conclude that for the sizes studied in our experiments, there is no additional overhead coming from task management.

4.3 Conclusion

This chapter presents the architecture and the primary evaluation of our framework, showing that it is fully operational, on homogeneous scenarios of a naive implementation of blocked matrix multiplication.

The entry point regarding the connectivity of the FPGA in the computing node is the PCIe bus. We choose RIFFA as an accessing framework that provides a software and a hardware interface of the bus, splitting the connection in a number of individual bi-directional channels that can be driven concurrently.

From the software point of view, we have extended the StarPU runtime system, creating a dedicated driver responsible for driving FPGA devices in a transparent way, similar to the software counterparts. On top of RIFFA, we have built Conor a software library managing the allocation of the device channels as well as the data exchange and provides performance results corresponding to the behavior of every transaction. On the hardware side, we have designed a connector that operates as a low latency intermediate between a RIFFA channel and a hardware task.

The evaluation of the integration is based on a blocked version of matrix multiplication, where every task computed a block of the result matrix. This series of experiments is conducted on homogeneous scenarios. On every run, tasks are running either on software or on hardware, using the standard eager scheduling policy. For the experiments we create tasks of different granularity exposing that the overhead imposed by the data movement is not always compensated in cases where the corresponding amount of execution load is not sufficient. For the tasks of small granularity, the performance per task is similar among the software and the hardware case. For the computationally heavier task, the acceleration obtained by the massive parallelism of the FPGA, is enough to overcome the overhead of the data transmission, getting a hardware task that outperforms its software counterpart. The results show that our framework does not impose any significant overhead to the execution of the application on any scenario.

Chapter 5

Heterogeneous Scheduling

Contents

This chapter introduces our development towards heterogeneous execution scenarios involving concurrent task execution on CPU cores and the FPGA. Chapter 4 presents the design and the architecture of our framework, as well as an evaluation of: a) the performance overhead that comes with the FPGA b) the platforms overall programmability. The evaluation of that chapter is based on a series of homogeneous experiments, where tasks are implementing for two different target processing units, the CPU and the FPGA, but on every run they are executed only on one of the two. Moreover, the complexity of the application under test in terms of its dependencies is very limited; said differently tasks are completely independent and can all execute concurrently. The target of this part of the work is to allow the execution of more complex applications in heterogeneous scenarios, where there are dependencies between tasks, that impose priorities, and tasks are executed on different processing units during the run time. Our reference application for this scenario is the Cholesky decomposition. StarPU comes with an existing implementation of Cholesky decomposition, with task implementations for the CPU and the GPU.

In our experiments, we chose one of the tasks assembling Cholesky decomposition, the *GEMM* kernel, optimized it and tried to provide heterogeneous execution scenarios where *GEMM* runs on the FPGA while the rest of the tasks run on the CPU. *Gemm* was chosen since it is the task executed the most during the decomposition, as shown in 5.3. The first step to accomplish the aforementioned goal is to ensure that with our integration we can actually obtain heterogeneous execution scenarios. The second step is to provide a hardware implementation of *GEMM*, that can deliver enough performance to compensate the overhead of the movement between the host memory and the device of its input and output dependencies. The third step is to provide a performance model of the FPGA implementation of *GEMM* and run a series of experiments to observe the effect of different scheduling policies on the result performance.

For the accomplishment of the first step, we conducted a series of experiments on the previous blocked matrix multiplication application, exploring the behaviors of our framework on a truly heterogeneous execution scenario. This series of experiments is presented in section 5.1. The analysis of *GEMM* and the development process of a highly optimized hardware implementation is presented in section 5.4. Unfortunately, with respect to the time limits of this thesis, we did not manage to complete this series of experiments, leaving the performance model of *GEMM* and the scheduling analysis of Cholesky decomposition on the perspectives of this study.

5.1 Heterogeneous Matrix Multiplication

This section describes a series of heterogeneous experiments we conducted in order to verify the ability of our framework to deliver truly heterogeneous executions. Heterogeneity here corresponds to the concept of concurrent task execution on resources of different architectures, in this case the CPU and the FPGA.

To demonstrate cases of heterogeneous execution, we used the blocked version of matrix multiplication of chapter 4, allowing a task to be executed either on a CPU core (using the *cpu_mult* function), or on one of the available hardware tasks at FPGA (using the *fpga_mult* function. That being said, for this series of experiments we used the version of our framework, where data exchange happens explicitly. We applied a random scheduling policy, where a task can be randomly chosen to execute in any of the two available processing units.

On the FPGA side there were four available hardware tasks corresponding to the multiplication operation. The node was equipped with four CPU cores, allowing a

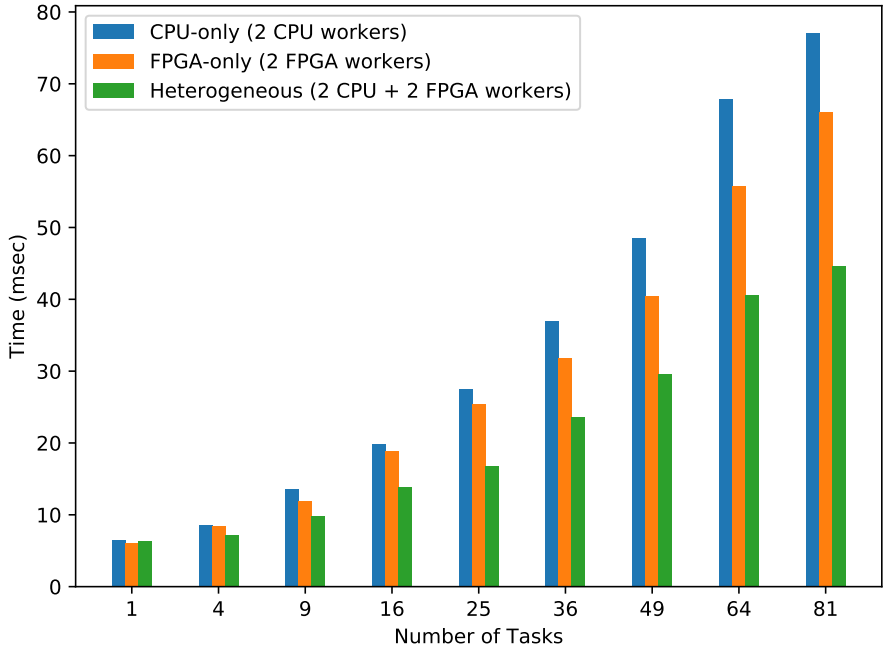
maximum of four workers in total. The FPGA configuration here was identical to the one used for the experiments of the previous chapters. Since every processing unit needed to be represented by a worker thread on the software side and since there should be one to one mapping between CPU cores and worker threads, we limited ourselves to four worker threads in total in order to allow heterogeneous execution. Half of the threads were associated to a CPU core as a working unit and the other half were associated to a hardware task on the FPGA.

Our experiments were conducted on the same fashion as the homogeneous equivalents of chapter 4. We run a series of executions, determining the task size (orthogonal blocks of dimension size equal to 64 or 256), as well as the number of tasks per execution. The number of tasks was determined by specifying the number of partitions per dimension as well as the size of every dimension of the corresponding matrix (figure 4.5).

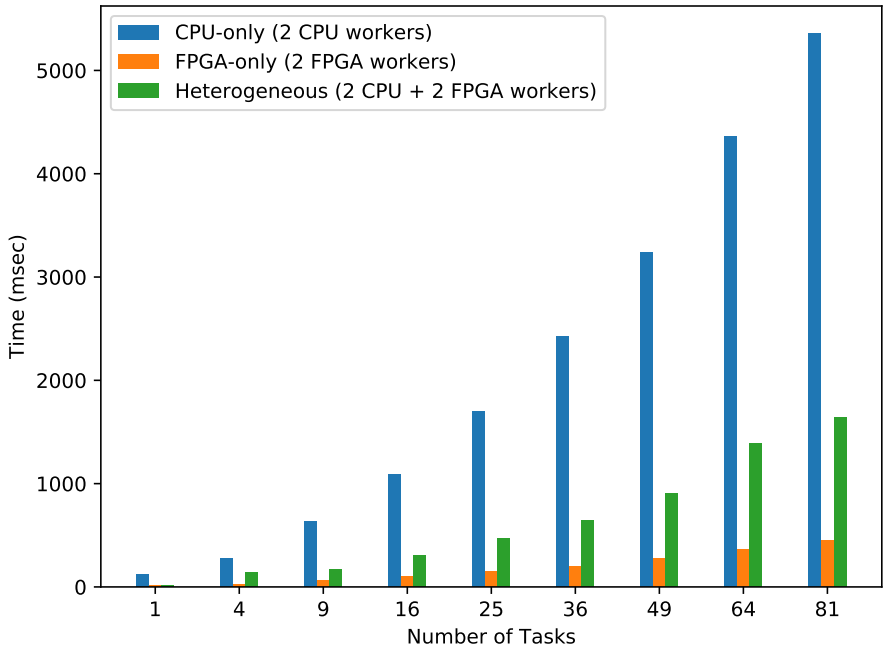
Figure 5.1 shows a comparison based on the total execution time of matrix multiplication for the equivalent matrix sizes between the two homogeneous and the heterogeneous scenario. For a fair comparison, on the homogeneous scenarios, we limited the number of workers of every unit to two. The performance per task remains the same as expected, to the one presented in section 4.2.3.

Figure 5.1a shows the comparison between the total execution time for the light weight tasks (orthogonal blocks of dimension equal to 64). The CPU and the FPGA implementation of a task at this granularity show similar performance. Figure 5.1b shows the comparison between the total execution time for the heavy weight tasks (orthogonal blocks of dimension size equal to 256). In this case, the execution time of the FPGA implementation of a task is faster compared to the one of the CPU implementation.

In this series of experiments a task could either execute on the CPU or the FPGA. Tasks were executing concurrently in both of the resources. The results were marked with the "Heterogeneous" label, and were compared with the equivalent homogeneous scenarios of execution, were tasks were executing either exclusively on the CPU or exclusively on the FPGA.



(a) Results on the low-weight scenario — Orthogonal 64 × 64 Blocks



(b) Results on the heavy-weight scenario — Orthogonal 256 × 256 Blocks

Figure 5.1: Performance results on heterogeneous execution scenarios of blocked matrix multiplication for block sizes of 64x64 (figure 5.1a) and 256x256 (figure 5.1b)

With this series of experiments we provided a proof of concept, of execution scenarios where a part of the application is running on the CPU while another is running

on the FPGA. We observed that even with a single type of tasks, scheduling can have a strong impact on performance.

In the low-weight tasks, performance per task for the CPU and the FPGA case was similar, and we saw that even by applying a simple random policy, we could obtain better total execution time with the combination of the two. In the heavy-weight scenario, things change. The performance per task between the CPU and the FPGA case are not comparable. Just by assigning a part of the workload to the CPU slows-down the overall performance, compared to a pure FPGA execution. Nevertheless, the results we obtained here, can not be generalized for more sophisticated scheduling policies.

5.2 Cholesky Decomposition

Cholesky decomposition or Cholesky factorization, is a numerical method that, for a Hermitian positive-definite matrix A, finds a lower triangular matrix L such that:

$$A = L * L^H$$

A matrix X is called Hermitian if and only if:

- X is square
- $X = X^H$, where X^H is the conjugate transpose of matrix X.

A matrix X is called positive-definite if and only if:

- X is symmetric
- For every non-zero column vector z, the product $z^T * X * z$ is positive, where z^T is the transpose of z

A matrix X is called lower triangular if and only if every element above its main diagonal is zero.

The conjugate transpose of a real matrix X is equal to its transpose:

$$X^H = X^T, \ when \ X \ is \ real$$

Since we only focus on real matrices, the factorization we are focusing on is the one below:

$$A = L * L^T,$$

where A is a real Hermitian positive-definite matrix, and L a real lower triangular matrix.

Cholesky decomposition is a method widely used in a wide range of applications from the numerical solution of partial differential equations, to Kalman filters (algorithms to estimate unknown variables of a complex system out a series of measurements observed over time via their joint probability distribution) and Monte Carlo simulations (method used for simulating systems with multiple correlated variables).

Cholesky decomposition is known to be a good candidate to demonstrate the benefit of scheduling in heterogeneous environments. Hence it has been thoroughly studied in platforms assembled by a combination of CPUs and GPUs [9]. The application is composed of tasks of different kinds that are known to adapt more or less efficiently among processing units of different architecture.

5.3 Analysis On The Implementation Of The Cholesky Decomposition.

Most efficient software implementations of Cholesky decomposition (like the one of BLAS -3) are typically assembled by four kernels operating on blocks of the matrix A : *POTRF*, *TRSM*, *SYRK* and *GEMM*. Listing 11 shows a C-based pseudo-code of the core of the application.

The aforementioned kernels are presented below:

- *POTRF* is a software kernel that implements directly the Cholesky decomposition. It is an efficient way to decompose small blocks, and hence it is applied directly to the diagonal blocks.

- *TRSM* is a triangular matrix equation solver. In its general form, it solves the equation $M * X = alpha * N$, where M is a triangular matrix. The kernel in this context is applied to $M = A[k][k]$ and $N = A[i][k]$. The result matrix X, overwrites N, in this case $A[i][k]$.

- *SYRK* is a kernel that performs a rank-k matrix-matrix operation between a symmetric matrix N and a matrix M of the form $N := alpha * A * A' + beta * N$. In this context is applied to $N = A[i][i]$ and $M = A[i][k]$. As indicated by the mathematical formula, matrix N is updated with the new values that derive from the computation.

- *GEMM* is a generic matrix multiplication solver that can be mathematically formed as $N = alpha * op(M_1) * op(M_2) + beta * N$ and is thoroughly analyzed in section 5.4. In this context $N = A[i][j]$, $M_1 = A[i][k]$ and $M_2 = A[j][k]$.

```
1  for( k=0; k<N; k++){
2      DPOTRF(RW, A[k][k]);
3      for( i=k+1; i<N; i++){
4          DTRSM(RW, A[i][k], R, A[k][k]);
5      }
6      for( i=k+1; i<N; i++){
7          DSYRK(RW, A[i][i], R, A[i][k]);
8          for( j=k+1; j<i; j++){
9              DGEMM(RW, A[i][j], R, A[i][k], R, A[j][k]);
10         }
11     }
12 }
```

Listing 11: C-based pseudo-code of the core of Cholesky decomposition.

Code 11 shows the core of Cholesky decomposition. Figure A.1 of appendix A shows a graphical representation of the task graph of Cholesky decomposition for N=4. Following the code of listing 11, the algorithm iterates over every diagonal block. So, on the k^{th} iteration, *POTRF* is applied on block $A[k][k]$. Then a *TRSM* is applied on every block of the same column bellow the diagonal block, this is every block $A[i][k]$, where $i > k$. A *SYRK* kernel is applied on every block diagonal block after k (i.e. every block $A[i][i]$, where $i > k$). And a *GEMM* kernel is applied on every block between the diagonal and the k^{th} column, this is every block $A[i][j]$, where $i, j > k$.

No *TRSM* kernel can execute before the completion of the *POTRF* on block $A[k][k]$. No *SYRK* kernel can execute on block $A[i][i]$ before the completion of *TRSM* on block $A[i][k]$. In other words, no *SYRK* kernel can execute on a diagonal block before the completion of the *TRSM* kernel of the same row. Lastly, no *GEMM* kernel can execute on block $A[i][j]$ before the completion of *TRSM* on blocks $A[i][k]$ and $A[j][k]$.

5.4 Developing A Hardware Design Of GEMM

General Matrix Multiply - *GEMM*- is a kernel that performs matrix multiplication of two matrices, although in a more general way than the *mmult* examined in chapter 4. The mathematical formula associated to the kernel is the one below:

$$C := alpha * op(A) * op(B) + beta * C$$

,where *alpha* and *beta* are constants, A is a x by z matrix, B is a z by y matrix, C is a x by y matrix and

$$op(X) = \begin{cases} X & \text{,transX = 'N'} \\ X^T & \text{,transX = 'T'} \\ X^H & \text{,transX = 'H'} \end{cases} \tag{5.1}$$

indicates the order that data are stored in memory. Our experiments only focus on real matrices where:

$$X^H = X^T$$

.

Figure 5.2 exposes an example of how a 3 by 3 matrix X is stored in memory when *transx* is either 'N' (figure 5.2b, line X) or 'T' (figure 5.2b, line X^T).

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

(a) The original 3 by 3 matrix X

Address	0	1	2	3	4	5	6	7	8
X	1	2	3	4	5	6	7	8	9
X^T	1	4	7	2	5	8	3	6	9

(b) Memory mappings of X and X^T

Figure 5.2: Demonstration of memory mapping of a 3 by 3 matrix X and its transpose X^T

In the CPU implementation of the *GEMM* kernel, different cases are distinguished according to the order the input matrices are stored in memory (whether they are stored in a transpose way or not). For the FPGA implementation such thing is not necessary, since data are transferred from the host to the device memory, and hence they can be remapped. Indeed for matrices A and B, *transA* and *transB* are transmitted before the elements themselves, and if they are originally stored in a transpose fashion, they are re-arranged in the FPGA memory to a non-transpose mapping.

After this data re-arrangement, the FPGA version of the kernel becomes:

$$C = alpha * A * B + beta * C$$

.

5.4.1 Hardware Design Optimizations Of GEMM.

IO template: The very first step for writing a hardware task compatible to the HEAVEN framework is the definition of its input, output, and control ports. Listing 12 shows the definition of the wrapping function of the *GEMM* hardware task named *top*. Following the template requirements of chapter 4, *top* has one 64-bits wide input port - *instream*, one 64-bits wide output port - *outstream* and a 32-bits wide control port - *control*. Every port is set to implement an *ap_hs* protocol (listing 12 lines 3-5). Lastly, there is no control signal set for the completion of the tasks execution.

```
1 void top(hls::stream<DATA_WIDTH> &instream,
  ↪  hls::stream<DATA_WIDTH> &outstream, hls::stream<int> &control)
2 {
3 #pragma HLS INTERFACE ap_hs port=instream
4 #pragma HLS INTERFACE ap_hs port=control
5 #pragma HLS INTERFACE ap_hs port=outstream
6 #pragma HLS INTERFACE ap_ctrl_none port=return
```

Listing 12: The high level function implementing *GEMM* has three arguments of type *hls::stream*. One input (*instream*) and two outputs (*outstream, control*). On the hardware level, streams are implemented as simple handshake interfaces.

Receiving Dependencies: In chapter 4 we specified that the dependencies of a task need to be sent in a specific order. Conor worker will first send the *STARPU_R* and then the *STARPU_RW* dependencies of a task, in the order they are registered during the codelet defintion and task creation. For the up-streaming phase, it is the *STARPU_RW* dependencies expected according to their declaration order, and then the *STARPU_W* ones.

Respecting the aforementioned restrictions the hardware task expects first to receive matrix A, then matrix B, and last matrix C since there is both and input and an output dependency on the result matrix. Listing 13 shows the code associated to the hardware side declaration and receiving of matrix A. It is exactly the same procedure for matrix B. The control branch of lines 9 and 18 ensure the non transpose remapping of the matrix to the FPGA memory.

HLS ARRAY_RESHAPE is a memory optimization technique provided by Vivado HLS, that performs data partitioning and mapping to the BRAM. By default, when an array is defined, Vivado HLS maps it automatically to a BRAM block. A BRAM block can be either single or dual ported allowing a total maximum of two

memory operations per cycle. This memory throughput would not allow to exploit the available parallelism of the device. *HLS ARRAY_PARTITION* would allow every row or column of a matrix to be mapped to a different BRAM block increasing proportionally the number of memory operations per cycle. This partitioning strategy by default could lead to a bad memory usage. *HLS ARRAY_RESHAPE* is a combination of partitioning and mapping, that maps several (in our case 2) rows or columns of the matrix to the same BRAM block, without decreasing the throughput of memory operations.

```
    REAL A[DIM][DIM];
#pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    int Ar, Ac;

    *((DATA_WIDTH *) temp)= instream.read();
    Ar = temp[0];
    Ac = temp[1];

    if(TA==0){
        IO_Ar_NT:for(int i=0; i<DIM; i++){
            IO_Ac_NT:for(int j=0; j<DIM; j+=2){
                *((DATA_WIDTH *) temp)= instream.read();
                A[i][j] = temp[0];
                A[i][j+1] = temp[1];
            }
        }

    }else{
        IO_Ar_TR:for(int i=0; i<DIM; i++){
            IO_Ac_TR:for(int j=0; j<DIM; j+=2){
                *((DATA_WIDTH *) temp)= instream.read();
                A[j][i] = temp[0];
                A[j+1][i] = temp[1];
            }
        }
    }
```

Listing 13: Defining, Receiving and Remapping the input arrays A and B

Matrix C is the last matrix to be received as mentioned previously. Listing 14 shows the implementation of the receiving procedure. *HLS ARRAY_RESHAPE* is

also applied on the C matrix to allow the increased memory throughput during the writing phase of the computation.

An important performance optimization at this point is the overlapping of communication and part of the computation. As shown in listing 14, instead of storing C itself, we store $beta * C$ while receiving the elements of the matrix.

Last performance optimization at this phase of the kernel is pipelining the receiving loop (Listing 14, line 11), with an initiation interval of 3 cycles (initiation interval refers to the number of cycles needed to complete the first iteration of the body of the loop).

```
1      //Get C, C = BETA * C;
2      REAL C[DIM][DIM];
3  #pragma HLS ARRAY_RESHAPE variable=C complete dim=2
4      int Cr, Cc;
5
6      *((DATA_WIDTH *) temp)= instream.read();
7      Cr = temp[0];
8      Cc = temp[1];
9      IO_Cr:for(int i=0; i<DIM; i++){
10         IO_Cc:for(int j=0; j<DIM; j+=2){
11 #pragma HLS PIPELINE II=3
12             *((DATA_WIDTH *) temp)= instream.read();
13             C[i][j] = BETA * temp[0];
14             C[i][j+1] = BETA * temp[1];
15         }
16     }
```

Listing 14: Defining, Receiving array C. Overlap communication with the computation of the $C = beta * C$ part of the algorithm

The computation's phase: Listing 15 shows the main phase of computations of the kernels that corresponds to the $C = alpha * A * B$ part. It corresponds to a 3-level nested loop. The optimizations techniques we used on this part is a combination of *loop pipelining* and *loop unrolling*.

Loop unrolling flattens the iterations of the loop to a certain factor, allowing the execution of the entire unrolled load in one clock cycle, if and only if the memory throughput is sufficient to feed the computation. This hardware optimization comes at a cost in terms of resource utilization, since the number of logic elements utilized for the implementation of the unrolling is proportional to the unrolling factor. This

indicates the necessity of applying the *HLS ARRAY_RESHAPE* optimization that was applied on the A, B and C blocks.

Loop unrolling was applied to the inner loop (Listing 15, line 7, loop C_i) using the *HLS UNROLL* directive. Without specifying an unrolling factor, the loop was unrolled completely. The same technique was applied to the middle loop (Listing 15, line 3, loop C_m). This time the loop was unrolled to a factor of 2, because of resource constraints as well as a bottleneck imposed by the memory's throughput. Along with *loop unrolling*, *loop pipelining* with an initiation interval of 5 cycles was also applied to C_m. This initiation interval was decided in order to obtain higher target frequencies.

```
1       //COMPUTATIONS
2       C_o:for(int i=0; i<DIM; i++){
3           C_m:for(int k=0; k<DIM; k++){
4   #pragma HLS PIPELINE II=5
5   #pragma HLS UNROLL factor=2
6               REAL A_PART = ALPHA * A[i][k];
7               C_i:for(int j=0; j<DIM; j++){
8   #pragma HLS UNROLL
9                       C[i][j] += A_PART *B[k][j];
10              }
11          }
12      }
```

Listing 15: The core of the computation which corresponds to the $alpha * A * B$ part.

Return result: The last phase of the task, is the phase of data write back, end more precisely the up-streaming of the C- matrix. In chapter 4 is specified that before data up-streaming, the size of the exchange should be propagated through the control channel (Listing 16, line 2). Without the use of any directive, the data exchange happens at a pace of one 64-bits word per cycle, assuming that the consumer can sustain this throughput, which is the optimal exchange performance we can expect in this context.

```
1       //WRITE OUTPUT
2       control.write(Cr*Cc);
3
4       OI_Cr:for(int i=0; i<DIM; i++){
5           OI_Cc:for(int j=0; j<DIM; j+=2){
6               temp[0] = C[i][j];
7               temp[1] = C[i][j+1];
8               outstream.write( *((DATA_WIDTH *) (temp)) );
9           }
10      }
```

Listing 16: Writing back the result matrix C.

5.4.2 Hardware GEMM - Performance Evaluation

The following chapter presents a performance analysis of the FPGA version of *GEMM*, based on the evaluation of the hardware design presented in Section 5.4.1. The analysis follows the optimization process, where we were trying to obtain the maximum performance for the given underlying resources. The results presented in the chapter, are taken from the static analysis performed at the end of synthesis by Vivado HLS.

The target frequency for all the configurations of this series of experiments is set to 250MHz, corresponding to a period of 4ns. The tool (Vivado HLS) managed to deliver the designs at a period of 3.41ns, which corresponds to a maximum frequency of 293MHz.

The performance results of table 5.1 correspond to the performance we could obtain with the optimization techniques of section 5.4.1 on kernels of various sizes. We focused on orthogonal blocks, whose dimension size was 256, 512 or 1024; the last being the largest design that fits into the area of our FPGA chip.

Performance \ Dimension size	256	512	1024
Latency (cycles)	558875	2231835	8920091
Throughput (Giga Operations per second)	15.6	25	55

Table 5.1: Timing results corresponding to the complete kernel (computations plus IO).

The estimation on the latency of the kernels or the operations per cycle are based on the assumption that data are available from the software side without any delay. In

other words, to achieve this performance, the software side should be able to feed the hardware tasks at a throughput of two words per cycle, and receive the upstream-ed data at the same speed.

Tables 5.2 and 5.3 show the resource demands of the corresponding designs in absolute and relative values respectively. Table 5.2 is useful in order to understand the amount of resources required in order to obtain a kernel of the tested sizes at the indicated performance, but does not help to understand the relative size of a kernel compared to the size of the FPGA chip. On the other hand, table 5.3 provides a comparative representation between the kernels of different sizes and the amount of the available resources.

From table 5.3 one can obtain the number of kernels of every size that can be synthesized concurrently on a single design, with respect to the available resources of our FPGA. For the case where every dimension is of 256, the limiting factor is the LUT (27% for a single kernel), allowing a maximum of three such kernels to be printed on a single design concurrently. For the cases of dimension size 512 and 1024 the limiting factor is the available amount of BRAM blocks; resulting hardware designs of a single 512 wide kernel or a single 1024 kernel.

Resource type \ Dimension size	256	512	1024
BRAM_18K	570	1140	2733
DSP48E	520	1032	2056
FF	95749	222781	438351
LUT	120718	120845	120970

Table 5.2: Resource usage per complete kernel (computations plus IO)

Resource type(%) \ Dimension size	256	512	1024
BRAM_18K	19	38	92
DSP48E	14	28	57
FF	11	25	50
LUT	27	27	27

Table 5.3: Resource usage per complete kernel (computations plus IO) as a percentage of the total amount of resources of the FPGA chip

As already stated, the aforementioned results correspond to the estimation of Vivado HLS. Despite the fact that in theory we should be able to obtain the expected designs, Vivado was not able to pass the synthesis steps at the frequency of 250Mhz

for any of the previous kernels. Since both Vivado HLS and VIVADO are proprietary tools from Xilinx, we have no way to further investigate the reason behind this incompatibility between the estimation of Vivado HLS and the synthesis failure of VIVADO. The only complete design (kernel plus connector plus RIFFA backend) that passed the synthesis step, is a single 256-wide kernel at the frequency of 100Mhz. Assuming that no better frequency can be obtained by the vendors tools, we can further investigate the effect of different Vivado HLS directives, that will allow us to obtain the same performance at lower resource demands.

5.5 Discussion

This second chapter of contribution is dedicated to the work conducted in order to evaluate the behavior of our framework to heterogeneous execution scenarios. Section 5.1 presents execution scenarios where tasks of the blocked Matrix Multiplication example of Chapter 4 are executed both in the CPU and the FPGA concurrently. The results of this series of experiments on the one hand prove that our framework can be used for truly heterogeneous execution scenarios, involving processing units of different types along with the FPGA. On top of that, we observed that even for parallel applications with trivial dependencies, the impact of heterogeneity and scheduling can be significant.

The initiative for the rest of this work, is to prepare the ground for more complex experiments, showing the impact of sophisticated scheduling policies in such cases. Here applications are composed of tasks of different characteristics ,their dependency resolution is not trivial, and they are known to be sensitive to different scheduling approaches. Cholesky decomposition is a typical such application; it is composed by tasks of different type that are known to be more or less suitable for processing units of different kinds.

In this chapter, we present the fundamental tasks assembling Cholesky decomposition. For our experiments, we chose *GEMM*, since it is the most heavily used task within a round of execution. We exposed the steps and optimization techniques employed in order to obtain a fairly optimized hardware design of *GEMM*, along with the associated performance resource utilization estimation.

Unfortunately, we did not manage to obtain the desired execution scenarios within this study, since there is still a number of engineering trade-offs that need to be resolved considering our hardware implementation of *GEMM*. Those trade-offs come

from the fact that the heuristics of the High Level Synthesis tool regarding the sustainable frequency of the design did not match with the synthesis abilities of the back-end during the implementation of the design.

In the perspectives of this work, we plan to complete our heterogeneous experiment, by creating the performance model associated with our hardware implementation and applying different scheduling policies.

Chapter 6

Conclusion and Future Work

Contents

6.1 Conclusion

This work was conducted in order to provide a runtime system that supports the execution of parallel applications on heterogeneous resources that involve FPGA.

Chapter 4 presented the first part of contribution of this work, the design and implementation of the framework that can facilitate this heterogeneous execution, as well as a study of the performance of the framework on homogeneous execution scenarios of a basic blocked matrix multiplication application.

The entry point regarding the connectivity of the FPGA in the computing node is the PCIe bus. We chose RIFFA as an accessing framework that provides a software and a hardware interface of the bus, splitting the connection in a number of individual bi-directional channels that can be driven concurrently. From the software point of view, we extended the StarPU runtime system, creating a driver responsible for driving FPGA devices in a transparent way, similar to the software counterparts. On top of RIFFA, we built Conor, a software library managing the allocation of the device channel as well as the data exchange and provides performance results corresponding

to the behavior of every transaction. On the hardware side, we designed a connector that operates as a low latency intermediate between a RIFFA channel and a hardware task.

The evaluation of the integration was based on a blocked version of matrix multiplication, where every task computed a block of the result matrix. This series of experiments were conducted on homogeneous scenarios. On every run, tasks were running either on software or on hardware, using the standard *eager* scheduling policy. For experimental purposes, we created tasks of different granularities exposing that the overhead imposed by the data movement is not always compensated in cases where the corresponding amount of execution load is not sufficient. For the tasks of small granularity, the performance per task was similar among the software and the hardware case. For the computationally heavier tasks, the acceleration obtained by the massive parallelism of the FPGA was enough to overcome the overhead of the data transmission, getting a hardware task that outperformed its software counterpart. The results showed that our framework did not impose any significant overhead to the execution of the application on any scenario.

Chapter 5 presented an evaluation of the matrix multiplication application of Chapter 4 for truly heterogeneous scenarios, where tasks were executed both in the CPU and the FPGA concurrently. This series of experiments, at first, was a proof of concept that our environment is fully operational on heterogeneous execution scenarios. On top of that, we saw the strong impact of heterogeneity and scheduling, even for simple applications like blocked matrix multiplication. Distributing the computational load between the CPU and the FPGA, can increase the overall performance when the performance of the two is equivalent, but can also slow down the final performance in cases where the two implementations differ significantly.

The second part of this chapter was dedicated to describing the steps followed in order to obtain an execution scenario of a more complex application. Indeed, both the execution scenarios of chapters 4 and 5 were based on an application with trivial dependencies and complexity, that did not allow the power of sophisticated scheduling policies to be expressed. The motivation and goal was to observe the impact of scheduling on applications with complex execution footprint, where the dependency resolution is not trivial. Such applications are composed of tasks of different characteristics and are known to be sensitive to different scheduling policies. Our reference application to study the behavior of our framework in such scenario was the Cholesky decomposition. In this chapter, one could find the fundamental task types assembling the Cholesky decomposition. For our heterogeneous experimentation, we

chose *GEMM*, since it is the most heavily used task within a round of execution. We exposed the steps and optimization techniques employed in order to obtain a fairly optimized hardware design of *GEMM*, along with the associated performance resource utilization estimation.

6.2 Future Work

This section covers the perspectives of this work towards an execution environment that can provide augmented performance of our reference platform architecture, as well as better understanding of the behavior of heterogeneous execution and its correlation with scheduling. The main performance bottleneck that can be directly overcome, is the upgrade of the interconnection technology between the host and the FPGA (Section 6.2.2). A more complex research direction, is the utilization of the on-board memory of the FPGA, that will allow better data reuse by decreasing the amount of data moving between the accelerator and the host memory (Section 6.2.5). The aspect of heterogeneous scheduling has been merely exploited within this work, but the provided infrastructure can host more complex heterogeneous experiments, that can expose the benefits of efficient workload mapping on heterogeneous parallel applications with complicated dependencies (Section 6.2.1). For a further exploration of the potentials of such a framework, a fruitful analysis is around the scalability of the platform, conducting experiments with more than one FPGA operating concurrently (Section 6.2.4).

6.2.1 Heterogeneous Experiments

The main motivation behind this work is to provide a framework that allows applications to utilize efficiently heterogeneous resources involving FPGA. The power of such an approach, compared to the existing device accessing libraries presented in Chapters 2, 3 comes from:

- The automatic management of low level mechanisms needed to ensure the correct execution of a parallel architecture in such platforms
- The scheduling policies that are provided, or that can easily be ported, that allow an efficient mapping of the workload to the underlying resources with respect to some criteria (performance, energy consumption, etc).

The benefits of the first point have been thoroughly demonstrated and analyzed within this work, more precisely in Chapters 3 and 4. The second point remains to be explored, since with this work, all the necessary tools for such study are available.

Given the experiments obtained by the development needed for this work, the main challenge of such studies is to identify the workload that, when executed in hardware, can show enough speed-up to overcome the overhead imposed by the data exchange between the host and the FPGA. Along with the identification of the workload, it is also challenging to choose the proper optimization strategies that will lead to the optimal hardware design, especially for developers with background in software engineering.

The most immediate results of the effect of heterogeneous scheduling on a fairly complex application can be obtained for the case of the Cholesky decomposition. In Chapter 5, we presented a series of steps towards an execution scenario were part of the application was running on the CPU (or any other accelerator of the platform) and the *GEMM* kernel on the FPGA. We went through the optimization strategies and we managed to obtain a fairly optimized hardware design of the kernel. However, there are still some design decisions to be made regarding the trade-off between the frequency of the design and the granularity of the hardware kernel. Once those parameters are set, we need to develop a performance model for this new architecture, that will be used as an heuristic of the schedulers presented in section 5. Once the aforementioned prerequisites are set, the environment is set for the execution of the application using different scheduling policies.

Hopefully, the optimizations conducted on *GEMM* will be enough to overcome the overhead imposed by the data exchange, and combined with different scheduling policies, we will obtain an overall performance increase for the entire application.

6.2.2 Interconnect Version Upgrade

A significant overhead on the use of accelerators is the penalty imposed by the data exchange associated to the execution of the load. Although this penalty can not be completely eliminated since data need to be transferred between the device and the host, current technology allows to reduce significantly its effect.

In our framework, we used the first generation of PCIe interface to communicate with the device, because the bandwidth was enough to feed the number of kernels we were concurrently using at the frequency of 250Mhz. PCIe generations 1 and 2 use a data encoding with 20% overhead to the theoretical bandwidth being lost just there. Taking this overhead into account, the per-lane bandwidth of a single lane for the first

generation of PCIe is 250MB/s for every direction, while for Gen 2 the bandwidth becomes double (500MB/s). In their modern FPGA boards, Xilinx for example provides PCIe support for the third and fourth generation of connection which comes at much higher bandwidth. To explore the full potential of such communication speed, we should re-design fundamentally the architecture of the connection, since twelve channels operating at 250Mhz cannot make use of the full capacity of such links, even when we double the bus width of every channel.

Reducing the absolute communication overhead should allow the data aware scheduling policies to offload more work to the FPGA, that will hopefully increase the overall performance. It will also allow to consider hyper-threaded solutions, where more than one Conor worker will be mapped to one physical core, and investigate if this could bring any performance improvement.

6.2.3 Dynamic Reconfiguration

In the scope of this thesis, the FPGA designs are static. This means that the developer needs to decide how many instances of every kernel will be actually implemented and configured to the FPGA. This imposes an upper limit regarding the absolute maximum number of hardware kernels that can be executed on a single configuration (and consequently on a single run).

An extension of our framework with a capability to reconfigure dynamically parts of the FPGA would bring great flexibility to the overall design. In such case, several implementations of several tasks can be provided in hardware, and configured on the fly to the device when the scheduler decides to. The only responsibility of the designer will be to design hardware kernels that will fit to the amount of resources dedicated to the reconfigurable area.

In order for such an option to be feasible, there is a series of extensions that needs to take place in the runtime part of the existing framework, besides the hardware oriented upgrades. First of all, Conor needs to be extended to be able to keep track of the state of every device at every given time of the execution. Secondly, StarPU and Conor need to collaborate, so that the first will be able to issue a reconfiguration request to the second, ensuring the corresponding hardware task is currently idle (i.e., there is no Conor worker executing the hardware task that had been previously configured to the device). Lastly, since the reconfiguration time of some reasonably heavy hardware tasks can be long, smart scheduling policies could be employed. One could imagine to overlap reconfiguration with execution for example, or to trigger the reconfiguration process ahead-of-time if the hardware task is idle.

6.2.4 Multi-Board Experiments

One of the main requirements that our framework satisfies is the ability to host more than one FPGA devices simultaneously. This distinguishes our approach from similar ones that are focused on SoC architectures where a single FPGA is attached to a multicore processor.

The experimental set-ups of chapters 4 and 5 show the strong dependency of the final performance to the amount of underlying resources. Both set-ups try to explore the capabilities of FPGA, without reaching the full potential of the theoretical computational ceiling. In a realistic scenario, multiple tasks of an application will be synthesized and finally executed in hardware (FPGA). Having multiple connected devices will allow such a configuration without imposing a great bottleneck on the amount of resources allocated to every task; a limitation that would lead to poor performance.

Given the current state of our framework, this next step is straightforward. On a platform with multiple FPGA devices attached, the programmer needs to decide the which kernels will be executed on which device, create the hardware designs, and provide the configurations to StarPU, before the execution, the same way this happends with single-board experiments. Then Conor and StarPU can transparently handle the allocation and managements of the channels associated to the communication with every kind of hardware task.

6.2.5 Use On-Board DDR

The background of this work (Chapter 2) presented the different types of memory that are associated to an FPGA. Unlike the BRAM blocks, that are tightly coupled with logic elements and scattered all across the fabric, the on-board memory is external to the fabric, but it can be accessed in a way similar to how main memory is accessed by the CPU. There is a difference of orders of magnitudes between the capacities of the two. In the case of our board (VC709), the total amount of the available BRAM that can be found in the chip is around 50MB, while the on-board memory can be up to 8 GB.

Extending our framework to support the on-board memory would allow:

- to handle kernels of bigger granularity, when the available memory was the limiting design factor

- to dramatically reduce the amount of exchanged data, since data blocks that reside on the on-board memory can be re-used by hardware tasks, as long as they are valid.

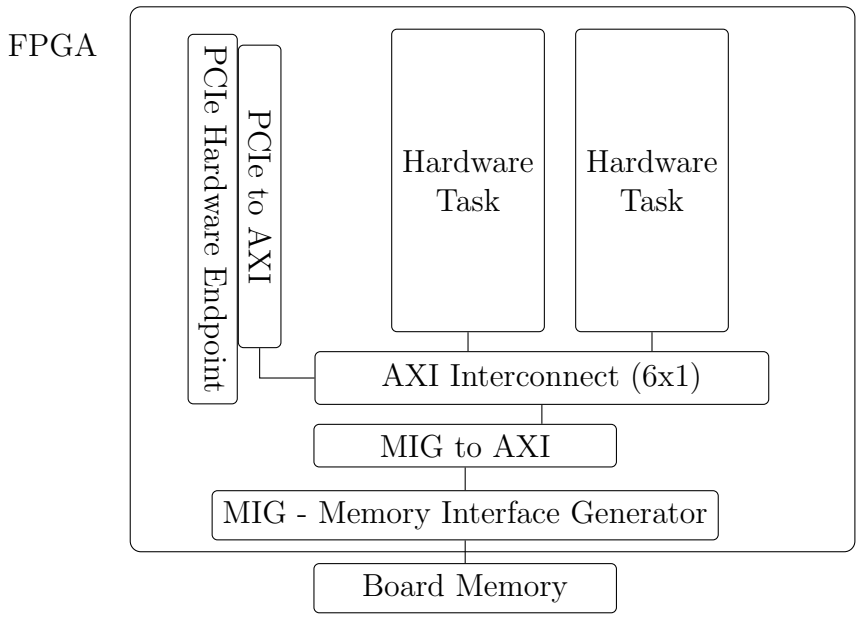


Figure 6.1: A design of an extended version of the framework able to exploit the on-board FPGA memory.

For such an extension, there is a series of architectural changes both on the software and hardware design of the proposed framework, that are presented in figure 6.1. On the hardware side, a memory controller needs to be used in order to access the on-board memory from the FPGA, on top of which an AXI2MIG interface can be implemented, that will allow the hardware tasks to access the on-board memory via an AXI bus. The host can communicate with the on-board memory directly through an AXI bus, imposing a PCI2AXI interface on top of the PCIe bus. On the software side, Conor needs to be extended with a primitive memory allocator. Moreover, the mechanism via which *STARPU_CONOR_MEM* remains consistent needs to be updated, since there is no need to always transfer data to and from the device.

Appendix A

Appendix

```
int _starpu_conor_driver_run_once(struct _starpu_worker
 ↪  *conor_worker)
{
    unsigned memnode = conor_worker->memory_node;
    int workerid = conor_worker->workerid;

    _STARPU_TRACE_START_PROGRESS(memnode);
    _starpu_datawizard_progress(memnode, 1);
    if (memnode != STARPU_MAIN_RAM){
        _starpu_datawizard_progress(STARPU_MAIN_RAM, 1);
        }
    _STARPU_TRACE_END_PROGRESS(memnode);

    task = _starpu_get_worker_task(conor_worker, workerid,
     ↪  memnode);

    job = _starpu_get_job_associated_to_task(task);

    /* can CONOR  perform that task ? */
    if (!_STARPU_CONOR_MAY_PERFORM(job))
    {
        /* put it and the end of the queue ... XXX */
        _starpu_push_task_to_workers(task);
        return 0;
    }

    struct starpu_perfmodel_arch* perf_arch =
     ↪  &conor_worker->perf_arch;

    _starpu_set_current_task(job->task);
    conor_worker->current_task = job->task;

    res = execute_job_on_conor(job, task, conor_worker, rank,
     ↪  perf_arch);
}
```

Listing 17: C-based pseudo-code of the core of the FPGA worker

```
struct starpu_perfmodel
{
    enum starpu_perfmodel_type type;

    double (*cost_function)(struct starpu_task *, unsigned nimpl);
    double (*arch_cost_function)(struct starpu_task *, struct
     ↪  starpu_perfmodel_arch * arch, unsigned nimpl);

    size_t (*size_base)(struct starpu_task *, unsigned nimpl);
    uint32_t (*footprint)(struct starpu_task *);

    const char *symbol;

    unsigned is_loaded;
    unsigned benchmarking;
    unsigned is_init;

    starpu_perfmodel_state_t state;
};
```

Listing 18: C-based pseudo-code of a StarPU performance model

```
#define PERTURBATE(a)     ((starpu_drand48()*2.0f*(AMPL) + 1.0f - (AMPL))*(a))

double cpu_chol_task_gemm_cost(struct starpu_task *task,
    struct starpu_perfmodel_arch* arch, unsigned nimpl)
{
    uint32_t n;

    n = starpu_matrix_get_nx(task->handles[0]);

    double cost = (((double)(n)*n*n)/50.0f/10.75/8.0760);
    return PERTURBATE(cost);
}

double cuda_chol_task_gemm_cost(struct starpu_task *task,
    struct starpu_perfmodel_arch* arch, unsigned nimpl)
{
    uint32_t n;

    n = starpu_matrix_get_nx(task->handles[0]);

    double cost = (((double)(n)*n*n)/50.0f/10.75/76.30666);
    return PERTURBATE(cost);
}
```

Listing 19: Example of a StarPU performance model. The code corresponds to the performance model of *GEMM* for the CPU and the GPU used in Cholesky decomposition.

```
module addbit(
a     ,//first input
b     ,//second input
ci     ,//curry input
sum     ,//sum output
co     //carry output
);
//In-Out Declarations
input a;
wire a;
input b;
wire b;
input ci;
wire ci;
output sum;
wire sum;
output co;
wire co;
//CODE
assign {co,sum}=a+b+ci;
//CODE_END
endmodule
```

Listing 20: An adder moduler written in verilog

```
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB)
{
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element that is
    // computed by the thread
    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        float elementA = A[ty * wA + k];
        float elementB = B[k * wB + tx];
        value += elementA * elementB;
    }

    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wA + tx] = value;
}
```

Listing 21: Device side code of matrix multiplication in OpenCL. The code computes the value of one element of the result matrix.

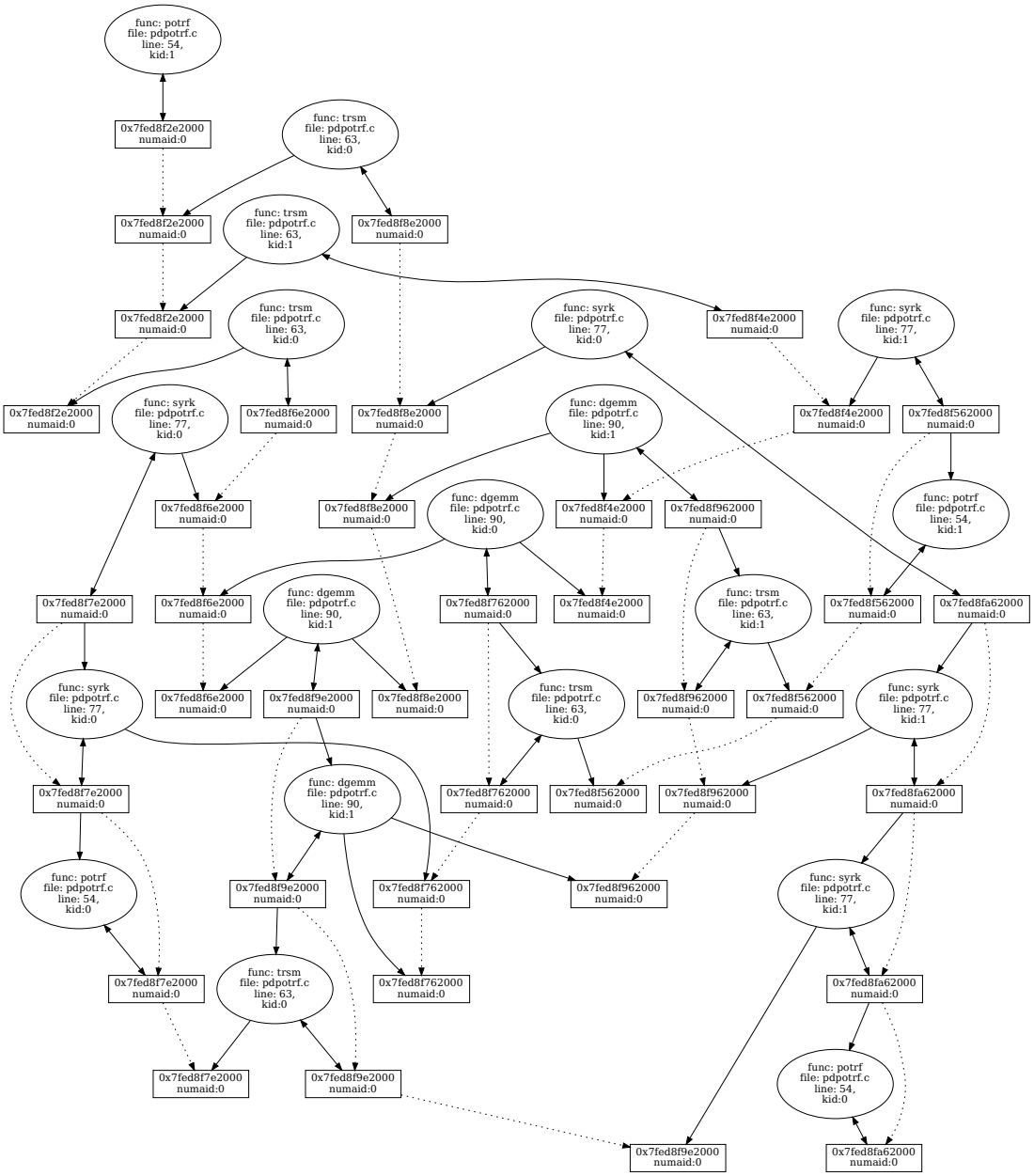


Figure A.1: Task Graph of Cholesky decomposition

Bibliography

[1] Clb architecture for the zynq7000 family of xilinx. URL https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[2] Presentation of intels goldmontplus architecture. URL https://www.extremetech.com/computing/261217-new-details-intels-goldmont-plus-cpu-architecture-inside-gemini-lake.

[3] Heterogeneous earliest finished time scheduler. URL https://en.wikipedia.org/wiki/Heterogeneous_Earliest_Finish_Time.

[4] Ioreg- a tool to display the kit registry. URL http://www.manpagez.com/man/8/ioreg/.

[5] Matrix multiplication in opencl. URL http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmopencl.

[6] Vc709, evaluation board for the virtex-7 fpga. URL https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf.

[7] Presentation of clb by xilinx. URL https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[8] Jason Agron and David Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 393–402. ACM, 2009.

[9] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically scheduled cholesky factorization on multicore architectures with gpu accelerators. 2010.

[10] Takayuki Akamine, Kenta Inakagata, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Reconfigurable out-of-order mechanism generator for unstructured grid computation in computational fluid dynamics. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 136–142. IEEE, 2012.

[11] P Alankrutha, H V Deepika, N Mangala, and N S C Babu. Multi-accelerator cluster runtime adaptation for enabling discrete concurrent-task applications. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 754–760. IEEE, 2014.

[12] Amazon. Amazon ec2 f1 instances, . URL https://aws.amazon.com/ec2/instance-types/f1/.

[13] Amazon. Amazon ec2 f1 instances, developers page, . URL https://github.com/aws/aws-fpga.

[14] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving programming model abstractions for reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):34–44, 2008.

[15] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving programming model abstractions for reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, 2008.

[16] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of . . . , 2006.

[17] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 228–235. IEEE, 2014.

[18] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing*, pages 863–874. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[20] Eduard Ayguadé, Rosa M Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco Igual, Daniel Jiménez-González, Jesús Labarta, et al. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[21] Ozalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 15–22. IEEE, 2002.

[22] Michael O. Bender and Michael O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems*, 35(3):289–304, 2002. ISSN 1432-4350. doi: 10.1007/s00224-002-1055-5.

[23] Brahim Betkaoui, David B Thomas, Wayne Luk, and Natasa Przulj. A framework for fpga acceleration of large graph problems: Graphlet counting case study. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8. IEEE, 2011.

[24] J C Beyer, E J Stotzer, A Hart, and B R de Supinski. OpenMP for accelerators. *OpenMP in the Petascale ...*, 2011.

[25] The OpenMP Architecture Review Board. The openmp specification version 4.0. *http://openmp.org/wp/openmp-specifications/*, 2013.

[26] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. Exploiting parallelism on gpus and fpgas with ompss. In *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, page 4. ACM, 2017.

[27] Vincent Boulos, Sylvain Huet, Vincent Fristot, Luc Salvo, and Dominique Houzet. Efficient implementation of data flow graphs on multi-gpu clusters. *Journal of Real-Time Image Processing*, (Special issue):online, October 2012. doi: 10.1007/s11554-012-0279-0. URL https://hal.archives-ouvertes.fr/hal-00746981.

[28] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568. IEEE, 2012.

[29] Jean-Philippe Chancelier, Bernard Lapeyre, and Jérôme Lelong. Using premia and nsp for constructing a risk management benchmark for testing parallel architecture. *Concurrency and Computation: Practice and Experience*, 26(9), 2012.

[30] Ying Chen, Tan Nguyen, Yao Chen, Swathi T Gurumani, Yun Liang, Kyle Rupnow, Jason Cong, Wen-Mei Hwu, and Deming Chen. Fcuda-hb: Hierarchical and scalable bus architecture generation on fpgas with the fcuda flow. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35 (12):2032–2045, 2016.

[31] Gregory Diamos and Sudhakar Yalamanchili. Speculative execution on multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[32] Gregory F Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200. ACM, 2008.

[33] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[34] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 161–170. ACM, 2014.

[35] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters ()*, 21(2): 173–193, 2011.

[36] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[37] Marie Durand, Francois Broquedis, Thierry Gautier, and Bruno Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 141–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[38] Fernando A Escobar, Xin Chang, and Carlos Valderrama. Suitability analysis of fpgas for heterogeneous platforms in hpc. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):600–612, 2016.

[39] Reza Rezaeian Farashahi, Bahram Rashidi, and Sayed Masoud Sayedi. Fpga based fast and high-throughput 2-slow retiming 128-bit aes encryption algorithm. *Microelectronics journal*, 45(8):1014–1025, 2014.

[40] Renato A Ferreira, Wagner Meira, Dorgival Guedes, Lúcia Maria de A Drummond, Bruno Coutinho, George Teodoro, Tulio Tavares, Renata Araujo, and Guilherme T Ferreira. Anthill: A scalable run-time environment for data mining applications. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pages 159–166. IEEE, 2005.

[41] Antonio Filgueras, Eduard Gil, Carlos Alvarez, Daniel Jimenez, Xavier Martorell, Jan Langer, and Juanjo Noguera. Heterogeneous tasking on smp/fpga socs: The case of ompss and the zynq. In *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*, pages 290–291. IEEE, 2013.

[42] Antonio Filgueras, Eduard Gil, Carlos Alvarez, Daniel Jimenez, Xavier Martorell, Jan Langer, and Juanjo Noguera. Heterogeneous tasking on smp/fpga socs: The case of ompss and the zynq. In *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 290–291. IEEE, 2013.

[43] Antonio Filgueras, Eduard Gil, Daniel Jiménez-González, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. OmpSs@Zynq all-programmable SoC ecosystem. In *the 2014 ACM/SIGDA international symposium*, pages 137–146, New York, New York, USA, 2014. ACM Press.

[44] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. Ompss@ zynq all-programmable soc ecosystem. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 137–146. ACM, 2014.

[45] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[46] Clément Foucher, Fabrice Muller, and Alain Giulieri. Exploring fpgas capability to host a hpc design. In *NORCHIP 2010*, pages 1–4. IEEE, 2010.

[47] Clément Foucher, Fabrice Muller, and Alain Giulieri. Online codesign on reconfigurable platform for parallel computing. *Microprocessors and Microsystems*, 37(4-5):482–493, 2013. ISSN 01419331. doi: 10.1016/j.micpro.2011.12.007. URL http://linkinghub.elsevier.com/retrieve/pii/S0141933111001293.

[48] Clément Foucher, Fabrice Muller, and Alain Giulieri. Online codesign on reconfigurable platform for parallel computing. *Microprocessors and Microsystems*, 37(4-5):482–493, 2013.

[49] Clément Foucher, Fabrice Muller, and Alain Giulieri. Online codesign on reconfigurable platform for parallel computing. *Microprocessors and Microsystems*, 37(4–5):482 – 493, 2013.

[50] The HSA Foundation. The hsa foundation specifications version 1.0. *http://www.hsafoundation.com/standards/*, 2012.

[51] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The Implementation of the Cilk-5 Multithreaded Language. *PLDI*, pages 212–223, 1998.

[52] Laurent Gantel, Amel Khiar, Benoit Miramond, Mohamed El Amine Benkhelifa, Lounis Kessal, Fabrice Lemonnier, and Jimmy Le Rhun. Enhancing reconfigurable platforms programmability for synchronous data-flow applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(3): 14, 2012.

[53] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[54] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE, 2013.

[55] Diana Göhringer and Jürgen Becker. FPGA-Based Runtime Adaptive Multiprocessor Approach for Embedded High Performance Computing Applications. *ISVLSI*, pages 477–478, 2010.

[56] Jose-Ernesto Gomez-Balderas and Dominique Houzet. A 3D reconstruction from real-time stereoscopic images using GPU. In *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP 2013)*, pages 253–258, Cagliari, Italy, October 2013. URL https://hal.archives-ouvertes.fr/hal-00878683. IEEE Xplore Compliant Files 979-10-92279-01-6.

[57] René Griessl, Meysam Peykanu, Jens Hagemeyer, Mario Porrmann, Stefan Krupop, Lars Kosmann, Patrick Knocke, Michał Kierzynka, and Ariel Oleksiak. FPGA-accelerated Heterogeneous Hyperscale Server Architecture for Next-Generation Compute Clusters. *First International Workshop on Heterogeneous High-performance Reconfigurable Computing*, 2015. URL http://h2rc.cse.sc.edu/h2rc-p12.pdf.

[58] Tsuyoshi Hamada, Khaled Benkrid, Keigo Nitadori, and Makoto Taiji. A comparative study on asic, fpgas, gpus and general purpose processors in the o (nˆ 2) gravitational n-body simulation. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 447–452. IEEE, 2009.

[59] A Hart. The OpenACC programming model. *Cray Exascale Research Initiative Europe*, 2012.

[60] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 456–469. ACM, 2016.

[61] Benjamin Humphries, Hansen Zhang, Jiayi Sheng, Raphael Landaverde, and Martin C Herbordt. 3d ffts on a single fpga. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 68–71. IEEE, 2014.

[62] Ra Inta, David John Bowman, and Susan M Scott. The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform. *Int. J. Reconfig. Comp. (IJRC) 2012*, 2012.

[63] Intel. Intel one api to rule them all is much needed. URL https://www.servethehome.com/intel-one-api-to-rule-them-all-is-much-needed/.

[64] Aws Ismail and Lesley Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 170–177. IEEE, 2011.

[65] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(4):22, 2015.

[66] Benedikt Janßen, Fynn Schwiegelshohn, Martijn Koedam, François Duhem, Leonard Masing, Stephan Werner, Christophe Huriaux, Antoine Courtay, Emilie Wheatley, Kees Goossens, et al. Designing applications for heterogeneous many-core architectures with the flextiles platform. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 254–261. IEEE, 2015.

[67] David H. Jones, Adam Powell, Christos Savvas Bouganis, and Peter Y K Cheung. GPU versus FPGA for high productivity computing. *Proceedings - 2010 International Conference on Field Programmable Logic and Applications, FPL 2010*, pages 119–124, 2010. ISSN 1946-1488. doi: 10.1109/FPL.2010.32.

[68] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. Citeseer, 1993.

[69] Lester Kalms and Diana Göhringer. Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[70] Md Ashfaquzzaman Khan, Matt Chiu, and Martin C Herbordt. Fpga-accelerated molecular dynamics. In *High-Performance Computing Using FPGAs*, pages 105–135. Springer, 2013.

[71] Lok-Won Kim, Sameh Asaad, and Ralph Linsker. A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(1): 5, 2014.

[72] Ryohei Kobayashi, Shinya Takamaeda-Yamazaki, and Kenji Kise. Towards a low-power accelerator of many fpgas for stencil computations. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pages 343–349. IEEE, 2012.

[73] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[74] Céline Labart and Jérôme Lelong. A parallel algorithm for solving bsdes. *Monte Carlo Methods Appl.*, 19(1), January 2013.

[75] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2003.

[76] Seyong Lee, Jungwon Kim, and Jeffrey S Vetter. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554. IEEE, 2016.

[77] Fabrice Lemonnier, Philippe Millet, Gabriel Marchesan Almeida, Michael Hübner, Jürgen Becker, Sébastien Pillement, Olivier Sentieys, Martijn Koedam, Shubhendu Sinha, Kees Goossens, et al. Towards future adaptive multiprocessor systems-on-chip: An innovative approach for flexible architectures. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 228–235. IEEE, 2012.

[78] João V F Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Maillard. Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. *Parallel Computing*, 44:37–52, 2015.

[79] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. High-throughput bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 73–82. ACM, 2010.

[80] Detailed list of authors on the respository. Starpu development repository. URL https://gforge.inria.fr/scm/?group_id=1570.

[81] Antonio Roldao Lopes and George A Constantinides. A high throughput fpga-based floating point conjugate gradient implementation. *Lecture Notes in Computer Science*, 4943(2008):75–86, 2008.

[82] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8, 2009.

[83] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, page 45, 2009. ISSN 10724451. doi: 10.1145/1669112.1669121. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-76749140917{&}partnerID=tZOtx3y1.

[84] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[85] Enno Lübbers and Marco Platzner. Reconos: An rtos supporting hard- and software threads. In Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *FPL*, pages 441–446. IEEE, 2007.

[86] Harris E Michail, GS Athanasiou, George Theodoridis, and Costas E Goutis. On the development of high-throughput and area-efficient multi-mode cryptographic hash designs in fpgas. *Integration, the VLSI Journal*, 47(4):387–407, 2014.

[87] Nicholas Moore, Miriam Leeser, and Laurie Smith King. VForce: An environment for portable applications on high performance systems with accelerators. *J. Parallel Distrib. Comput.*, 72(9):1144–1156, September 2012.

[88] Gerald R Morris and Khalid H Abed. Mapping a jacobi iterative solver onto a high-performance heterogeneous computer. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):85–91, 2013.

[89] Olivier Muller, Amer Baghdadi, and Michel Jézéquel. From parallelism levels to a multi-asip architecture for turbo decoding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(1):92–102, 2009.

[90] Kevin Nibbelink, Sanjay Rajopadhye, and Ross McConnell. 0/1 knapsack on hardware: A complete solution. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 160–167. IEEE, 2007.

[91] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue ()*, 6(2):40–53, 2008.

[92] NVIDIA. Drive px news, mar 2015. URL http://www.nvidia.com/object/drive-px.html.

[93] Nuno Oliveira and Pedro D Medeiros. A heterogeneous runtime environment for scientific desktop computing. In *International Conference on Vector and Parallel Processing*, pages 256–269. Springer, 2016.

[94] Nuno Oliveira and Pedro D Medeiros. Heterogeneous personal computing: A case study in materials science. *Procedia Computer Science*, 108:2398–2402, 2017.

[95] Francisco Ortega-Zamorano, José M Jerez, and Leonardo Franco. Fpga implementation of the c-mantec neural network constructive algorithm. *IEEE Transactions on Industrial Informatics*, 10(2):1154–1161, 2014.

[96] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218. IEEE, 2017.

[97] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

[98] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Efficient compilation of cuda kernels for high-performance computing on fpgas. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):25, 2013.

[99] Artur Podobas. Accelerating parallel computations with openmp-driven system-on-chip generation for fpgas. In *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 149–156. IEEE, 2014.

[100] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 60(1):79 – 93, 2014.

[101] Abid Rafique, George A Constantinides, and Nachiket Kapre. Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):24–34, 2015.

[102] B Ramkumar, AB Sinha, VA Saletore, and LV Kale. The charm parallel programming language and system: Part ii-the runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[103] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[104] Michael P Robson, Ronak Buch, and Laxmikant V Kale. Runtime coordinated heterogeneous tasks in charm++. In *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, pages 40–43. IEEE, 2016.

[105] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[106] Syam Sanal and J Pinalkumar. Multithreaded image processing using reconos on reconfigurable computing system. In *2018 International Conference on Emerging Trends and Innovations In Engineering And Technological Research (ICETIETR)*, pages 1–5. IEEE, 2018.

[107] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2014.

[108] Sean O Settle. High-performance dynamic programming on fpgas with opencl. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pages 1–6, 2013.

[109] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 117–126. ACM, 2014.

[110] Dezso Sima. The design space of register renaming techniques. *IEEE micro*, 20 (5):70–83, 2000.

[111] Hayden Kwok-Hay So and Robert W Brodersen. *Borph: An operating system for fpga-based reconfigurable computers*. University of California, Berkeley, 2007.

[112] Mostafa I Soliman and Ghada Y Abozaid. Fpga implementation and performance evaluation of a high throughput crypto coprocessor. *Journal of Parallel and Distributed Computing*, 71(8):1075–1084, 2011.

[113] Lukas Sommer, Jens Korinth, and Andreas Koch. Openmp device offloading to fpga accelerators. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 201–205. IEEE, 2017.

[114] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[115] P Sundararajan. High performance computing using FPGAs. *Xilinx White Paper: FPGAs*, 2010. URL http://china.zylinks.com/support/documentation/white{_}papers/wp375{_}HPC{_}Using{_}FPGAs.pdf.

[116] Yi-Gang Tai, Chia-Tien Dan Lo, and Kleanthis Psarris. Scalable matrix decompositions with multiple cores on fpgas. *Microprocessors and Microsystems*, 37(8):887–898, 2013.

[117] George Teodoro, Daniel Fireman, Dorgival Guedes, Wagner Meira Jr, and Renato Ferreira. Achieving multi-level parallelism in the filter-labeled stream programming model. In *2008 37th International Conference on Parallel Processing*, pages 287–294. IEEE, 2008.

[118] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin N Gurcan, Wagner Meira, Umit Catalyurek, and Renato Ferreira. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[119] TOP500. Top500 list of june 2018, ibm summit. URL https://www.top500.org/lists/2018/06/.

[120] Deepak Unnikrishnan, Sandesh Gubbi Virupaksha, Lekshmi Krishnan, Lixin Gao, and Russell Tessier. Accelerating iterative algorithms with asynchronous accumulative updates on fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 66–73. IEEE, 2013.

[121] B Sharat Chandra Varma, Kolin Paul, and M Balakrishnan. Accelerating 3d-fft using hard embedded blocks in fpgas. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 92–97. IEEE, 2013.

[122] R Vasudevan, Sathish S Vadhiyar, and Laxmikant V Kalé. G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 349–358. ACM, 2013.

[123] Philippe Virouleau, Pierrick Brunet, Francois Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *OpenMP in the Petascale Era*, pages 16–29. Springer International Publishing, Cham, 2014.

[124] Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.

[125] Lukasz Wesolowski. *An application programming interface for general purpose graphics processing units in an asynchronous runtime system.* PhD thesis, University of Illinois at Urbana-Champaign, 2008.

[126] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[127] Grant Wigley, David Kearney, and Mark Jasiunas. Reconfigme: a detailed implementation of an operating system for reconfigurable computing. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[128] Guiming Wu, Yong Dou, Junqing Sun, and Gregory D Peterson. A high performance and memory efficient lu decomposer on fpgas. *IEEE Transactions on Computers*, 61(3):366–378, 2012.

[129] Yan Xu, O. Muller, P. Horrein, and F. Petrot. Hcm: An abstraction layer for seamless programming of dpr fpga. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 583–586, Aug 2012.

[130] Depeng Yang, Gregory D Peterson, and Husheng Li. Compressed sensing and cholesky decomposition on fpgas and gpus. *Parallel Computing*, 38(8):421–437, 2012.

[131] Dong Yin, Ge Li, and Ke-di Huang. Scalable mapreduce framework on fpga accelerated commodity hardware. *Internet of Things, Smart Spaces, and Next Generation Networking*, pages 280–294, 2012.

[132] 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.