Efficient High Performance FPGA-Based Applications Design via SDAccel

BY

LORENZO DI TUCCI
B.S., Politecnico di Milano, Milan, Italy, July 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2016

Chicago, Illinois

Defense Committee:

Wenjing Rao, Chair and Advisor
Piotr J. Gmytrasiewicz
Marco D. Santambrogio, Politecnico di Milano

It's lack of faith that makes people afraid of meeting challenges, and I believed in myself.

Muhammad Ali

ACKNOWLEDGEMENTS

I believe this work is the result of many people and thanking them all would be impossible. Consciously or not, they helped me in reaching this achievement.

The first big thank goes to my family. Thank you very much for giving me such a fantastic opportunity of studying at UIC, thanks for the help and the constant support you give me. Thank you *cachi* for being my number one supporter, and for staying always by my side.

A big thank goes to my advisor, Marco D. Santambrogio for the trust, the opportunities, and the support in developing this work, your suggestions and guidance have been a key factor for the development of this work.

Thanks to Gianluca Durelli and all the people in the NecstLab at Politecnico di Milano, for the suggestions, for sharing their knowledge and for making the developing of this work more funny.

Thanks to all the guys at the Xilinx Research Lab in Dublin: Ken, Kimon, Zack, Peter, Corentin, Simon, Lisa, Baris and Dave for the help and the good time spent together. In particular, thanks to Michaela Blott for guiding me and giving me the opportunity to work in her team.

Thanks to all the people who shared this experience at UIC with me. We had a really good time in Chicago!

ACKNOWLEDGEMENTS (Continued)

Last but not least, I would like to thanks all the *Bargis' Lovers* for the awesome time spent together, for all the study hours and laughs. You have been my university family and really made these years a lot less heavy.

LDT

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMBA Advanced Microcontroller Bus Architecture. ix, 46

ANSI American National Standard Institute. ix, 40, 41

AOCL Altera OpenCL. ix, 57

API Application Programming Interface. ix, 52, 66, 69

ASIC Application Specific Integrated Circuit. ix, 35

AXI Advances eXtensible Interface. ix, 46

BPI Byte Peripheral Interface. ix, 100

BRAM Block Random Access Memory. ix, 12, 76

CAD Computer-Aided Design. ix, xiv, xv, 2, 59

CLB Configurable Logic Blocks. ix, 5, 7, 8

CPU Central Processing Unit. ix, 3, 23, 24, 59, 60, 67, 69–72, 107, 109

CSoC Configurable System on Chip. ix, 32

DDR Double Data Rate. ix, 56, 75, 99

DMA Direct Memory Access. ix

DSE Design Space Exploration. ix, 2

DSP Digital Signal Processor. ix, 7, 13, 35, 38

EDA Electronic Design Automation. ix, 35

EPROM Erasable Programming Read-Only Memory. ix, 6

FIFO First In First Out. ix, 34, 41, 57

LIST OF ABBREVIATIONS (Continued)

FPGA Field Programmable Gate Array. v, vi, viii, ix, xv, 1, 5–12, 21, 28–30, 32, 34, 35, 38, 40, 41, 43, 47, 49, 51–53, 56, 57, 59–64, 66, 68–72, 74, 75, 77–79, 90, 97–99, 101, 107

GCC Gnu Compiler Collection. ix, 41, 44, 47

GCUPS Giga Cell Update per Second. ix, 17, 25, 26, 81, 103–105, 111

GPCPU General Purpose Central Processing Unit. ix, 3

GPU Graphic Processing Unit. ix, 1, 21, 23, 59, 60, 62, 66, 67, 69–72, 75, 79

GUI Graphical User Interface. ix, 2, 30, 66, 71

HDF Hardware Design Flow. ix, xv, 10

HDL Hardware Description Language. ix, 2, 10, 11, 43, 44, 61

HLS High Level Synthesis. vi, ix, 2, 11, 12, 39, 43, 47, 49–51, 64, 66, 71, 73, 78, 90

HPC High Performance Computing. vi, ix, xv, 1, 3, 59, 60, 98, 101, 111

HW hardware. ix, xiv, 2

IDE Integrated Design Environment. ix, 50, 78

II Initiation Interval. ix, 95, 96

IOB Input Output Block. ix, 8

IP Intellectual Property. ix, 12, 49–51, 62

ISE Integrated Software Environment. ix, 46

JTAG Joint Test Action Group. ix, 100, 102

LLVM Low Level Virtual Machine. ix, 33

LP Low Power. ix, 39

LUT LookUp Table. ix, 7, 8, 12

LIST OF ABBREVIATIONS (Continued)

MAC Multiply-Accumulator. ix, 38

MCS Monte Carlo Simulation. ix, 27

MGT Multi-Gigabit Transceiver. ix, 99

OI Operational Intensity. ix, 14, 81

OS Operative System. ix, 69

PC Personal Computer. ix, 100

PCI Peripheral Component Interconnect. ix, 28

PCIe Peripheral Component Interconnect Express. ix, 52, 58, 69, 70, 97, 99

PMBUS Power Management BUS. ix, 100

QDR Quad Data Rate. ix, 56

RAM Random Access Memory. ix, 6, 7, 29

ROCCC Riverside Optimizing Configurable Computing Complier. v, ix, 32

ROM Read Only Memory. ix, 9

RTL Register Transfer Level. ix, 11, 12, 30, 31, 33, 35, 38, 39, 44, 46, 47, 49, 51, 72

SATA Serial Advanced Technology Attachment. ix, 99, 101

SDK Software Development Kit. vi, ix, 31, 51

SDRAM Synchronous Dynamic Random Access Memory. ix, 99

SODIMM Small Outline Dual In-line Memory Module. ix, 99, 101

SRAM Static Random Access Memory. ix, 6, 9

SW software. ix, xiv, 2

TCL Tool Command Language. ix, 37, 44, 49, 50, 71

LIST OF ABBREVIATIONS (Continued)

XML eXtensible Markup Language. ix, 46

SUMMARY

Custom hardware accelerators are widely used to improve the performance of software applications in terms of execution times and to reduce energy consumption. However, the realization of a hardware accelerator and its integration into the final system is a difficult and error prone process. For this reason, both industry and academy are continuously developing Computer Aided Design (CAD) tools to assist the designer in the development process. Although many of the steps of the design are now automated, system integration, SW/HW interfaces definition and drivers generation are still almost completely manual tasks. The latest tool by Xilinx however, aims at improving the hardware design experience by automating the majority of the steps in the design flow and by leveraging the OpenCL standard to enhance the overall productivity and to enable code portability.

This work provides an analysis and an overview of the new Xilinx SDAccel framework, comparing its design flow to other state of the art frameworks. In this context we use this tool to accelerate two case studies from the bioinformatics field. The first case study concerns *pairwise alignment* and the second one the *protein folding* problem. The work is organized as follows:

- We start with an introduction to our work, followed by a brief introduction to the context and our contributions in Chapter 1.

- Chapter 2 gives the reader an overview of Field Programmable Gate Arrays (FPGAs), followed by an introduction to the Hardware Design Flow (HDF). The chapter ends with a theoretical introduction of the two case studies that we developed.
- Chapter 3 describes some state of the art tools used in the design of hardware applications, comparing them and highlighting the main features of each.
- Chapter 4 analyzes the problem presented in this dissertation. It starts by describing the design for HPC and ends by talking of how new CAD tools aims at automating the steps of the hardware design flow.
- Chapter 5 introduces the tool we used to accelerate our case studies: *SDAccel*. It starts with an introduction to the framework, then it introduces its architecture and its main features. Finally, it discusses how we faced the problems of the tool and our contributions to its development.
- Chapter 6 describes how we decided to accelerate the two case studies introduced in Chapter 2. It explains the architectural choices that we've made, as well as the reasons that lead us to choose them.
- Chapter 7, presents the results of the two case studies, the experimental settings and the comparison of our result with state of the art implementation of the same algorithm.
- Finally, in Chapter 8 we draw the conclusions of this work and we provide some insights into possible future work.

CHAPTER 1

INTRODUCTION

In this chapter we want to provide an introduction to the matter discussed in this dissertation. More in particular, in Section 1.1 we present the context of our work, while Section 1.2 illustrates our contributions.

1.1 Context Definition

The performance requirements of computing systems are continuously increasing at all levels, from High Performance Computing (HPC) environments to mobile and embedded systems. It is becoming more and more clear that a software only solution is not a valid approach anymore. The large amount of data that the applications need to process, impacts massively on the performance. Furthermore, it has a great negative impact on the amount of energy that is need to perform the computation [1].

In this particular scenario, Hardware Accelerators have proven to be effective in the task of optimizing the ratio performance/power consumption of certain type of applications. One example of this can be the calculation of the Pearson coefficient as explained in [2]. By Hardware Acceleration we mean the process of using computer hardware to perform some task in a much faster way with relation to a pure software implementation. Hardware acceleration is usually performed with hardware accelerators such as Graphic Processing Units (GPUs) or FPGAs. The advantage of using such hardware acceleration is that normally, processors execute instructions

sequentially one at a time, by using a hardware acceleration it is possible to exploit *concurrency* and *parallelism*. The operation on a hardware accelerator can be executed in parallel, hence more than one instruction can be performed at the same time.

However, achieving the great benefits of hardware acceleration comes at a cost. In fact, the complexity of the process of designing the accelerator is very high. This complexity is well known and, over the years, both industry and academy have produced a wide range of CAD tools in order to assist the designer in the process of developing the HW accelerator [3–5]. These tools help the user by simplifying and automating the majority of the development phases. The designer, that had to code the description of the hardware by hand, is not strictly required anymore to manually write an Hardware Description Language (HDL) implementation of the HW core. Thanks to these simplifications, the designer can leverage High Level Synthesis (HLS) tools [3,6] to translate an high level code (generally C, C++ or SystemC) to HDL. The introduction of CAD tools allowed the complete automation of certain design step as HDL translation with HLS tools and application partition using Design Space Exploration (DSE) frameworks (mainly explored in academia), or the simplification of the design process by the assistance of a Graphical User Interface (GUI). However, even with the help and assistance of these tools, the designer is still required to perform manual integration of the Hardware Cores in the final architecture and he has to define all the HW/SW interfaces, program and implement the driver, write the control code to manage the core and then update the application to exploit the hardware accelerators. Manually performing these steps is an error prone task

and furthermore it requires the designer to have a deep knowledge of the system he has to optimize both at hardware and software levels.

1.2 Contributions

The contributions presented in this work, are the result of a 6-months internship at the Xilinx Inc Research Lab in Dublin, Ireland. During this period, I have been able to work with the SDAccel developing team and I had the task of performing benchmarking of HPC algorithms on Xilinx FPGAs and general purpose CPUs (GPCPUs).

My work also helped with the debugging of the tool itself as the bug I found were promptly solved by the development team.

This thesis, is the result of my intern period in Xilinx. The main contributions of this work is an analysis, compared to state of the art frameworks, of the SDAccel Design Flow. Note that, at the moment, the tool has not been released as it is still under development. Furthermore, will be presented an example of development using SDAccel on two different applications for the world of the bioinformatics. The first application concerns pairwise alignment of protein and is the Smith-Waterman algorithm. The second one is the Protein Folding, a process used to identify the tertiary structure of a protein.

Our results show the best ratio performance/power consumption, with relation to the considered state of the art implementations, for the Smith-Waterman algorithm. While for the Protein Folding algorithm, we have a final speed-up of 1.61x with relation to a pure software implementation. Note that we developed the first use case, in order to demonstrate the performance capabilities of the SDAccel framework while, for what concern the protein folding, we

wanted to demonstrate the improvement, in terms of design time, that the we gained by using the tool. It took us only 3 weeks, in fact, to have the results we are presenting here.

CHAPTER 2

BACKGROUND

This chapter presents the background of this work. We will start in Section 2.1, with an introduction to the hardware platform where we tested our two test cases, the Field Programmable Gate Arrays. Then, in Section 2.2, we will talk about how to design application for such devices by giving an overview of the Hardware Design Flow. Finally, in Section 2.3, we will introduce the two test cases used in this work.

2.1 FPGA

The Field Programmable Gate Array (FPGA) is a programmable device that is part of a particular family of integrated circuits that are used for custom hardware implementation. It is mainly composed of three parts: a set of logic blocks, CLB (or logic cells), an interconnection and a set of input/output cell (I/O Cells) that surround the device. FPGAs have been introduced by the company Xilinx in 1985. The user can program each of the three parts. A function that is implemented on such device, is firstly partitioned in different modules. For each module a logic block can be used to implement it. Once all the logic blocks have been implemented there is the needing to interconnect them all together. In order to do that, the interconnection logic need to be used.

Over the years, different kind of FPGAs have been produced. They mainly differentiate by the number of times you can program the device and from the technology that is used to

implement them. The technology has the goal of defining how the different blocks (input/output cells, logic cells and programmable interconnect) are physically realized. The three main technologies we are going to explain here are the Antifuse, SRAM and EPROM.

2.1.1 FPGA Technologies

TABLE I: OVERVIEW ON FPGA TECHNOLOGIES

Type	Technology	Pro	Cons
Antifuse	Antifuse	small area / low parasitic resistance	can't be reprogrammed
SRAM	Static RAM	reprogrammable	volatile
EPROM	Floating Gate	reprogrammable	hard to reprogram

Table I, shows the comparison between the three main technologies of FPGAs. The FPGAs based on the *antifuse* technology present the advantage of occupying less area, lower resistance and parasitic capacitance compared to transistors. The drawback is that they are *one-time programmable*. *SRAM*-based FPGAs are the most used ones. They can be programmed more than once, but require more area, and every time they are cycle-powered they need to be configured again. Finally, *EPROM* are based on floating-gate technology. They are reprogrammable, however the process for doing this is not very straightforward.

One of the main advantage of using an FPGA is in the fact that the hardware on the board is not fixed. Then if the user need to change the physical architecture of the function that he wants to accelerate, this is easily doable just by sending a new configuration file to the chip.

The user does not need to physically replace the circuit, allowing dynamic reconfiguration of the device. This is a very interesting factor, in particular for all the application, such as the one involving network appliances, where there is the needing of continuously update the system.

Now we will briefly discuss the general architecture of an FPGA. We will focus on FPGAs manufactured by Xilinx. However, the general principles discussed here are the same also for FPGA of others brands.

2.1.2 FPGA Architecture

As we said before, the FPGA is mainly composed of three main building blocks that are logic blocks, input/output blocks and interconnection logic (communication resources). All together, this three components create a reprogrammable infrastructure. However, FPGAs are composed of additional elements such as RAMs, DSPs and multiplexers. The elements that are physically implemented on the board are called *hard cores*. The programmer can leverage on them by integrating them with *soft cores*. This are the components that are instantiated by the user when the device is programmed.

2.1.2.1 Configurable Logic Blocks

The main component of an FPGA are the Configuration Logic Blocks or CLBs. They occupy the majority of the die area and are composed of one (or more) generators of functions that are realized with LookUp Tables (LUTs). The LUTs can be implemented using standard logic gates, multiplexers and latches and can implement any arbitrary logic function, according the their configuration. LUTs are surrounded by interconnection logic that aims at routing the signal to, and from, the LUT itself. When an FPGA is configured, the memory inside a

LUT is written and configured to perform the function required. Then, the interconnection is configured to route the signal correctly so that is possible to implement a complex function exploiting the rest of the logic on the board.

2.1.2.2 Input/Output Blocks

The I/O Blocks are needed in order to interconnect the internal logic and to route the signal from the internal logic to an output pin of the FPGA. Each I/O pin of the FPGA is mapped to only one IOB. During configuration, the memory of the IOB is written and information regarding the voltage and the direction of the communication are stored. It is possible, in fact, to establish monodirectional links as well as bidirectional ones.

2.1.2.3 Interconnection Resources

The communication resources are need in order to connect the CLBs and the IOBs together.

There are two main ways of performing this interconnection, respectively *Direct Interconnection* and *Segmented Interconnection*. In the former, the set of connection is spread all over the device, so that all the possible blocks are reachable

Logic blocks usually route the data on a channel that is chosen basing on the final destination of the signal. The main advantage of this kind of interconnection is in the fact that the the parasite resistance is more predictable as it is constant. The same thing for the capacitance. The latter, uses lines that need to be interconnected by means of programmable switch matrices and, also in this case, the connections cross the device in all its dimensions.

2.1.2.4 The Bitstream File

The bitstream file is the configuration binary that need to be *flashed* on the targeted FPGA in order to program it. It contains all the configuration information that will be stored in the configuration memory or in the ROM (Read-Only Memory) of the board. The ROM of the board can be used to solve the problem that every time a board is power cycled, the bitstream needs to be downloaded again. In fact, thanks to this memory, is possible to store the bitstream file and, when the board is power-cycled, load the values present in this memory, into the configuration memories. The bitstream file can be used either to program the full FPGA or only parts of it. In the second case there could be multiple partial bitstream that can be flashed in different times. As instance, an area of the FPGA can be reprogrammed while another area is computing. This configuration file can be flashed on the board using the dedicated interfaces that are physically implemented on the board. Among these there is the JTAG download cable and the SelectMap interface. The type of interface that is used depends mainly on the family of the FPGA that you want to program. In order to program the FPGA, they are usually equipped with some *configuration logic*. This logic has the task of understanding what is written in the binary file and to implement the interface that will manage the configuration data. There are also some *configuration registers* used to store the values given by the bitstream that will be passed to the SRAMs so that the device can be programmed.[7,8]

2.2 Hardware Design Flow

The Hardware Design Flow (HDF) is the process that a designer needs to follow in order to realize a hardware module that implements a specific task. The task could be an algorithm that is particularly compute intense and needs a hardware acceleration.

In order to perform a hardware acceleration of a task, the designer need to describe the algorithm by means of a *Hardware Description Language* (HDL).

The HDL is a language that is used to describe the hardware architecture of the component that we want to implement in hardware. There are two types of HDL, it can be *behavioral* or *structural*. If the designer writes the code in behavioral HDL, it means that he is focusing more on the behavior that he wants to implement. He will write what the task will have to do rather than how the system will look like. From the other hand, writing a structural HDL code, the designer will have to express what is the structure of the hardware module he wants to implement. Here, he will focus more on how many registers to implement and how to interconnect the various part of the system rather than writing what the algorithm will have to accomplish.

In this section, we will focus on the hardware design flow concerning Field Programmable Gate Array devices (FPGAs). FPGAs can be used as hardware accelerators by executing a particular task/algorithm (or part of it) of an application. In order to have the application running on the FPGA, the designer will have to follow multiple steps.

At the beginning, the application we want to accelerate is partitioned. This step is necessary in order to identify all the dependencies that are present between the different operations

of the algorithm. Identifying the dependencies, allow the programmer to understand what are the operations that can be executed in parallel. Obviously, to perform such partitioning the programmer needs to have a good knowledge of the algorithm. After understanding the algorithm, he should try to map the identified blocks to the computing resources of the board. [7] The hardware design flow can be seen as a two step operation that given a high level code produces as output a bitstream file that will be then downloaded to the FPGA board. Usually the first step is the translation of the high level code into HDL and then the implementation of the translated module into the target device. Nowadays, the translation into HDL code for particular domains is done with a process called *High Level Synthesis*. Hence, we can state that the two steps, presented here, of the hardware design flow are *High Level Synthesis* and *System Level Design*.

2.2.1 High Level Synthesis

Unfortunately, it is not easy to translate an algorithm into HDL. Furthermore, the development time of an algorithm with such language is high and not always justified. For this reason, many HLS tools have been created, with the aim of alleviating the programming of FPGAs. HLS tools attempt to describe an electronic architecture starting from a high level code and try to translate it to the hardware description language, without any manual intervention of the user. This facilitation makes the FPGA a more appetible device for programmers who do not have an hardware background and, as a consequence, raises the popularity of such devices. This step starts with an High Level code, usually expressed as C/C++ or SystemC, and outputs the RTL specification of our algorithm. This step is usually performed with a HLS tool as

Vivado HLS[3]. This kind of tool helps and guide the user in the creation of an IP Core, in fact, it has a compiler that allows the user to efficiently introduce code optimizations. For example, we can perform some unrolling, loop pipelining or partition variables (move array from BRAM to logic) by simply using the GUI of the tool. With Vivado HLS is also possible to test and benchmark our code both with a C simulation, as well as an RTL one, so that we can ensure the correctness of the circuit we are trying to synthesize. The result of the HLS is an IP Core, a description of the hardware circuit that implements the functionality of the algorithm that we want to accelerate by means of a Hardware Description Language that can be Verilog or VHDL.

2.2.2 System Level Design

After the creation of the IP Core, there is the System Level Design step. This step takes the IP Core of the previous step and allows the user to create all the rest of the hardware architecture needed to actually use the IP core, such as interconnection with input/output signals, memory, processors and so on.

Like in the previous phase, also during the System Level Design we can specify different implementations targeting the architecture that will be implemented on the platform. The Xilinx tool that can be used to do this step is called Vivado[4]. After the description of the architecture, Vivado takes care of the subsequent steps that involve the allocation of the resource, and the production of the final circuit by routing the resources together.

The result of this second phase is a bitstream file. This file contains all the configurations for a specific FPGA that specify the way the basics elements of the FPGA (e.g. LUT, BRAM,

DSP) need to be configured to implement the devised architecture. This configuration file needs to be flashed on the board so that, with the help of a control code, we can run our core on the platform.

2.3 Case Studies

Here we will provide an overview of the two test cases we developed for this work. First we will describe the *Berkeley Roofline Model*, then we will provide some information regarding the *Smith-Waterman* Algorithm; finally, we will talk of a second use case based on the *Protein Folding* problem.

2.3.1 The Roofline Model

With the advent of the multicore era, the conventional wisdom concerning computer architecture design faded and no new conventional wisdom raised yet. For this reason, the new microprocessors will be diverse. Some of these new architectures could exploit multi-threading, while others could replace the cache hierarchy with some explicit memory addressing. In such a context, where producers may create processors with a variable number of cores, there is the need for a model that provides the user with some guidelines regarding the performances of his application on the targeted system. The Roofline Model [9] is a model that tries to accomplish this task in an easy-to-understand manner.

This model, exploits the so called 'bound and bottleneck' analysis which, instead of trying to estimate the performance as stochastic models do, aims at providing more information regarding where are the bottlenecks of the application developed. Furthermore, it tries to give

quantification of what is affecting the performances of the system, so that the user knows where optimizations are needed.

The roofline model, relies on the fact that, in the near future, the majority of hardware architectures will have as main bottleneck the off-chip memory bandwidth, hence the roofline need to model the relation between the processor performance and the memory bandwidth.

The *Operational Intensity (OI)* is defined as the number of operation performed per each byte of DRAM traffic, i.e. the number of operation needed per each byte of information that goes into main memory after passing the whole cache hierarchy. Note that is not measured the traffic of data between processor and caches but between the latter and the memory.

The 2D chart, representing the roofline model, depicts the relation between floating-point performance, operational intensity and memory performance. Information regarding the floating-point performance can be retrieved by conducting some microbenchmark or by watching at the hardware specifications. Note that all the data retrieval needs to performed only once, even if the architecture is multicore.

An example of Roofline Model is observable from Figure 1. The graph is on a log-log scale. On the x-axis is represented the operational complexity in terms of number of Floating Point Operation per each Byte of data moved, while on the y-axis is shown the performance as million of floating point operations per second. The horizontal line, the one in light and darker red, represent the peak floating-point performance of the device analyzed. It is the hardware limit of the platform, so no application that run on such device, no matter how much is optimized, can have performance higher than the one of that line. The second line (light, and darker orange)

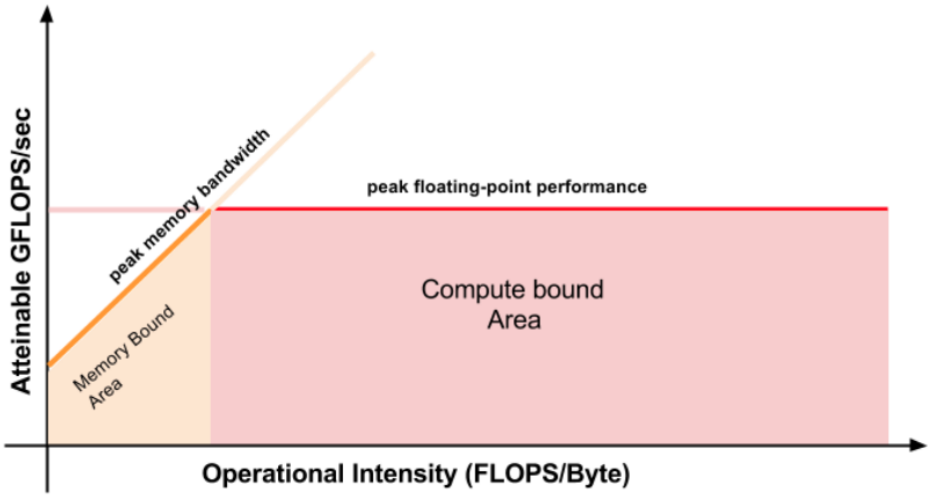


Figure 1: The Roofline Model

shows what is the maximum performance achievable, in terms of memory bandwidth, on the targeted platform. This two boundary lines, are ruled by the following formula:

$$GFLOPS = min\left\{ \frac{\left(^{PeakFloatingPoint}_{Performance}\right)}{\left(^{PeakMemory}_{Bandwidth}\right) \cdot \left(^{Operational}_{Intensity}\right)} \right.$$

The two lines, in Figure 1, intersect in a point that represents the maximum value of performance obtainable as it is represented by the peak memory bandwidth and the maximum value of computational performance. It is called the *ridge point*, and provide the programmer with important information. In fact, its x coordinate is the minimum operational intensity in order to achieve the best performances.

For each kernel the user wants to analyze, he has to start by identifying what is the value of operational intensity. Once this value is known, the user will search for the correspondence

on the x-axis. At this point there are two possibilities, the value can be either in the memory bound area or in the compute bound one. If it is in the memory bound area (highlighted in light orange in Figure 1), it means that the kernel is limited by the amount of bandwidth memory that is available on the system. Furthermore, in order to have more performance from the particular kernel, there should be more bandwidth memory. The right side of the chart, is the compute bound area. In this area, the kernel is limited by the compute resources available on the platform. An operational intensity that places itself in the compute bound area is said to be touching the roof. In fact, the two lines remember a roof and gives the name to the model.

The Roofline Model is created only once for the platform and hence can be used for different kernels. It also provides information of where to optimize a specific kernel. It is in fact possible to add multiple ceilings to the model so that the programmer will have information regarding the order of application of the optimizations. Each ceiling represent an optimization and is not possible to break the roof without performing the optimization. The programmer may solve this multiple ceiling in order, and this way he will have a precedence of optimizations to follow. Furthermore, an optimization that is in the ceiling over a previous one, can not be solved before solving the precedent.[9]

2.3.2 Smith-Waterman

The Smith-Waterman algorithm has been introduced firstly in[10] back in 1981 and it is based on the Needleman - Wunsch algorithm. It has been developed by T.F. Smith and M.S. Waterman and performs *local sequence alignment* between two strings that can be nucleotide or proteins. It is a *dynamic programming algorithm* as it decomposes a big problem into smaller

ones, storing the intermediate results and using it again when solving the next sub-problem. This algorithm is guaranteed to find the optimal local alignment between the two strings, called the *query* and the *database*, with relation to the scoring system that is being used.[11] Given a query N, a database M, a similarity system and an affine gap function, the algorithm calculates the *scoring matrix* S and the *TraceBack Matrix* T whose dimension is N x M each. Finally, it traces back the path of elements starting from the index identifying the maximum element in the Similarity matrix, and terminating whenever it finds a zero valued element. The performances of this algorithm are calculated as Giga Cell Update per Second, namely GCUPS. This measure the amount of cell update has been made in the amount of time, the formula to calculate the performance is:

$$GCUPS = \frac{N \times M}{ExecutionTime} \tag{2.1}$$

2.3.2.1 Step I : Matrices Calculation

The first step is the calculation of the similarity and the traceback matrix. During the calculation of the similarity matrix, there is the needing to keep track also of the index of the maximum element, as it will be necessary for the second step of the algorithm. The similarity matrix needs to be built so that

$$S(i, 0) = 0, 0 \leq i \leq m \tag{2.2}$$

and

$$S(0,i) = 0, 0 \leq i \leq n \tag{2.3}$$

where n and m are the size of the query and the database. Then, each element of the scoring matrix is populated following:

$$S(i,j) = \left\{ \begin{array}{ll} S(i-1,j-1) + s(N_i, M_j) & Match/Mismatch \\ max_{k \geq 1} S(i-k,j) + gap_{del} & Deletion \\ max_{k \geq 1} S(i,j-l) + gap_{ins} & Insertion \\ 0 & \end{array} \right\}, 1 \leq i \leq m, 1 \leq j \leq n \tag{2.4}$$

where

- $s(N, M)$ is a similarity function over the two strings.
- gap_{del} is the gap scoring value in case of deletion.
- gap_{ins} is the gap scoring value in case of insertion.

As it is observable from Figure 2, it is noticeable that each element depends on other three values: respectively the value that lies on the previous anti-diagonal, the value on its left and the value above. This means that, in order to compute the value considered, is necessary to compute the three previous values explained before.

The Similarity matrix is not the only matrix that need to be calculated at this stage. In fact, the algorithm calculates in parallel the TraceBack matrix. This second matrix, is used to

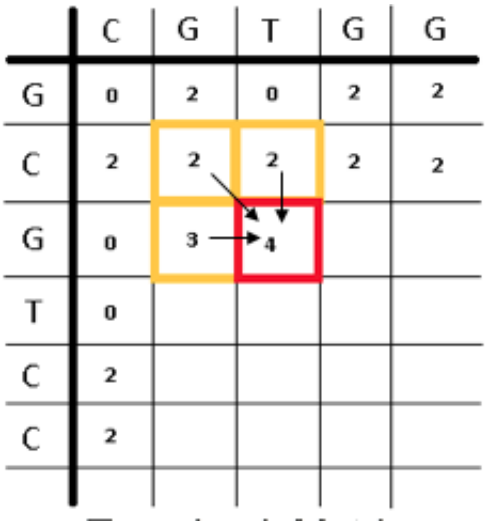


Figure 2: Step I : Similarity Matrix Calculation

store all the *directions* that the second step will will have to follow so that the final string can be calculated.

An example of TraceBack matrix is observable from Figure 3. As it understandable from eq. (Equation 2.4), in the similarity matrix we store the maximum value between 0 and other three values. The first one is a value composed of the sum of the element that lies on the anti-diagonal and another value that depends on the similarity function applied. The second element, is the sum of the value on the left of the one considered and the gap value in case of deletion. The third value is composed of the gap value in case of insertion and the element over the one considered. Once identified the maximum between this three values, the algorithm stores in the traceback matrix the position of the value that originated the identified maximum. As an example, in case the maximum value is the first one, the program will store in the traceback

Figure 3: Step I : TraceBack Matrix Calculation

matrix the value *north-west*, in the second case it will store *west* and in the third one *north*. In case none of the previous is the maximum, it means that the maximum value is a zero, then the program will store *center*.

At the end of the first step of the algorithm two matrices will be produced, the Similarity and the TraceBack Matrix.

2.3.2.2 Step II: Trace-Back

The final step of the algorithm, is the one that produces the real output: the best local alignment between the query and the database. This step, starts from the maximum index, identified in the previous step of the algorithm, and iteratively passes through the elements in the TraceBack matrix by following the directions stored (Figure 3). The process continues until a cell where the value stored is *center*. For each element in the traceback matrix, the

algorithm compare the query at that particular index with the database. In case of a match, it just rewrite the letters, in case there is a mismatch, if the direction is *north* the program inserts a - in the database string, while if the direction is *west* the - is inserted in the query string. Finally, the program outputs the two strings, expressing the best local alignment of the query with the database.

2.3.2.3 State of the Art

There are many implementation of the Smith-Waterman algorithm in the state of the art. However, is worth knowing that many of these architectures involve optimizations, as the use of heuristics, that reduce the execution time of the algorithm, at the cost of loosing precision. In fact, introducing heuristics in the algorithm, the Smith-Waterman is not guaranteed to find the optimal local alignment anymore. Examples of this are[12] and the FASTA[13]. Other implementations exploit pre-computed matrices that facilitate the calculation of the similarity matrix[14] or align more than one query to the same database string in parallel. Our implementation, from the other hand, is a pure Smith-Waterman guaranteed to find the optimal local alignment with respect to the score system used and align one query to one database. Here we will present some of the implementations found in the state of the art.

The implementations explored here, will be targeting GPUs[14–20] , co-processor as the Xeon-PHI[21] and FPGAs[22]. Companies, such as Convey[23], or Rivyera[24], developed clusters in order to support sequence alignment based on Smith-Waterman algorithm.

From the state of the art, is clear that there are many ways to parallelize this algorithm. The main challenges are related to the overall bandwidth of the system, the methodology used to parallelize the algorithm and how to manage memory limitation of the board.

In 1997, A. Wozniak proposed a Smith-Waterman architecture that exploited the parallelization along the anti-diagonal.[25] Wozniak found out that the anti-diagonals are dependency-free, hence he created an architecture that exploited the parallelism on the anti-diagonal. He benchmarked his architecture both using integer (32-bit) instructions, and VIS (Visual Instruction Set) on a ULTRA SPARC machine achieving overall performance of 18 MCUPS.

Instead of exploiting the vectorization of the anti-diagonal as in the previous section,[26] decided to use vectors of cell that are parallel to the query sequence (Figure 4). This implementation obviously has to deal with the dependencies among the different elements of the matrix. Note that in the previous implementation, the one proposed in[25], each anti-diagonal was dependency free. However, this implementation presents the advantage of being more simplified. Furthermore, the loading of data from memory is faster using this approach than by using the previous one. The implementation proposed in this paper, adds also other level of optimizations. This kind of enhancements are doable as the code they are implementing is not based on the original implementation proposed by Smith and Waterman but on the one proposed later by Gotoh in 1982 which involve the calculation of more matrices.

The Striped approach has been proposed by Michael Farrar in[27] and is based on the previous approach by Rognes and Seeberg. In this approach, the profile of the query, a step used in the Gotoh implementation of the Smith-Waterman algorithm, is performed using vectors paral-

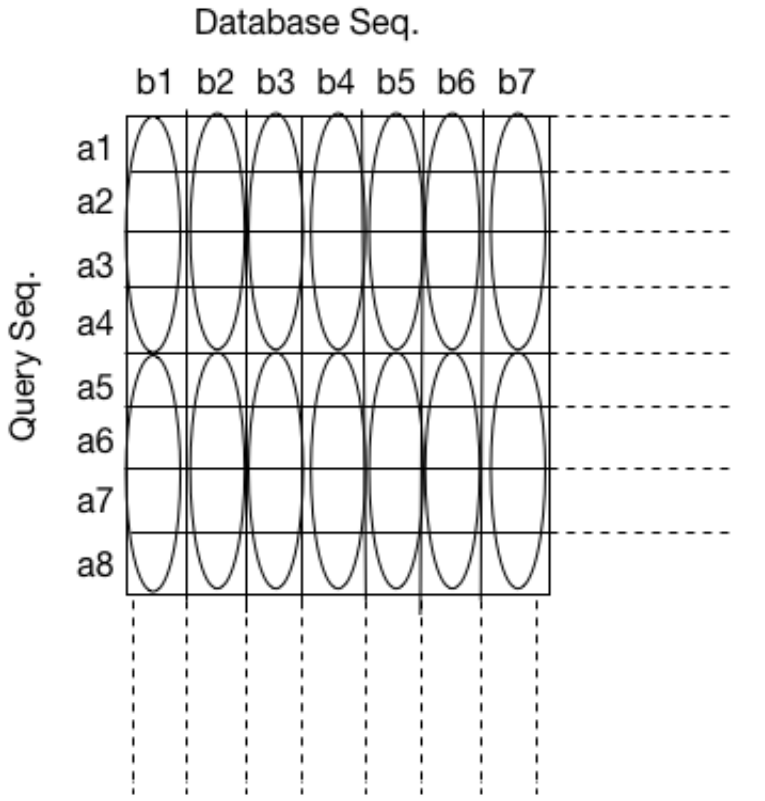


Figure 4: Vectors Parallel to Query Sequence

lel to the query string, but the query is split into equal segments so that the data dependencies are moved out of the inner loop of the algorithm.

Lukasz Ligowski and Witold Rudnicki in[15], suggest an implementation that process the similarity matrix horizontally. In this implementation, CPU and GPU operates concurrently; the host prepare the database for the GPU by organizing it in blocks of 256 sequences, then wait for the queries. Once the host receives the query, it run the algorithm on the GPU which

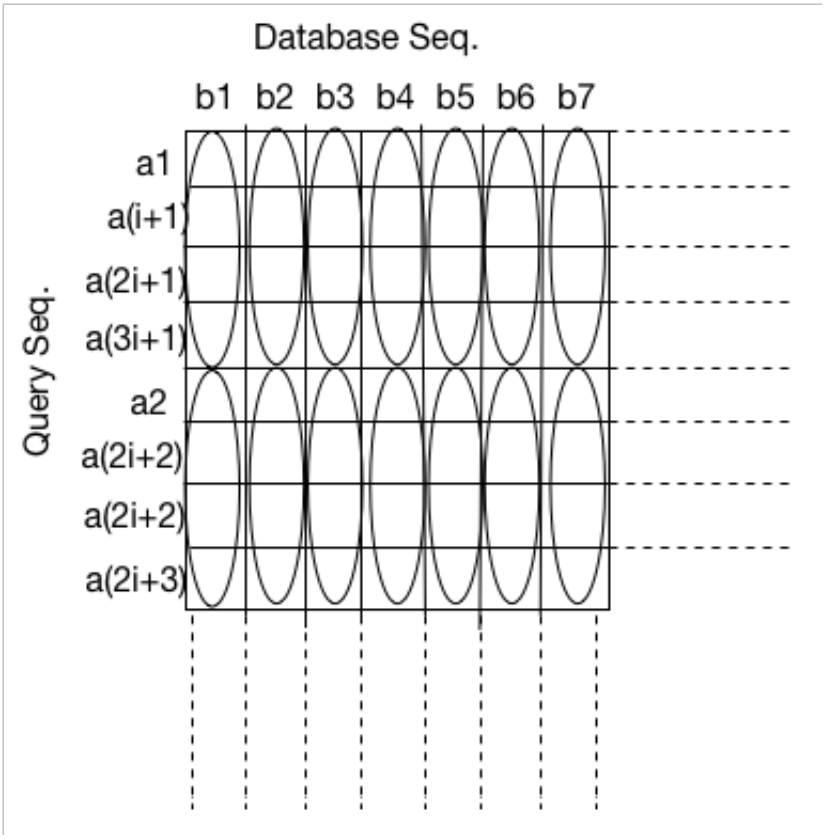


Figure 5: Striped Vectors Parallel to Query Sequence

return the maximum alignment score for each query processed. Then the CPU realigns the sequences and output the results. In this architecture, each thread process a single pair of sequences, in fact the query is shared by all threads. As said before, the similarity matrix is calculated horizontally with a sliding window of size 12. Furthermore, there is an outer loop that executes for 12 cell columns of the sliding window.

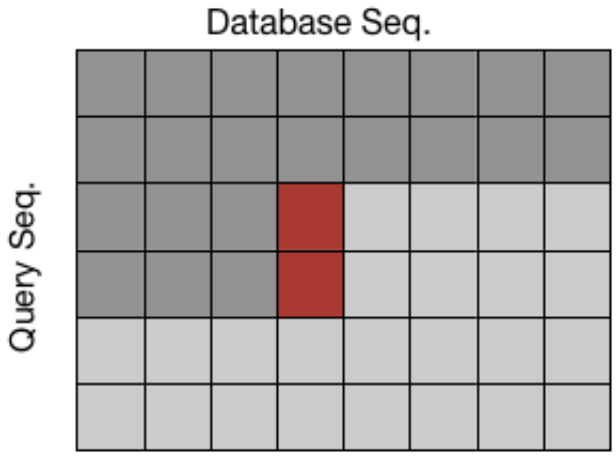


Figure 6: Horizontal Processing of Similarity Matrix

By using this implementation Ligowski et al. could theoretically achieve 93 GCUPS, however, due to the physical limitation of the target board (Nvidia 9800 GX2) they only achieve 14,5 GCUPS

A systolic array implementation is the one used by Sean O. Settle in [28]. A systolic array is a matrix of *processing elements*. Usually, each processing elements performs an operation and then passes the result of that operation to the next element in the matrix, that will perform another operation. In this way, the data will be passed from the beginning of the matrix and will go through all the processing elements, producing the final result of the computation.

In [28], the author creates a systolic array of processing elements where each PE (processing element) executes a *task*. A task is defined as an entity that executes a single OpenCL work-group containing only a single work-item. In this paper, a database sequence is streamed trough

the systolic array that is obtained by completely unrolling the inner loop of the smith-waterman algorithm. Then, private memories are used in order to have adjacent, diagonal, vertical and horizontal processing elements communicating. In the article the author suggests to store the sequence one element per byte, using the 4 least significant bits to store the 4 nucleic acids (A-C-G-T), and the 4 most significant to store control signals so that the sequences can be streamed efficiently through the systolic array.

The best performances achieved have been 24.7 GCUPS using a Nallatech PICEe-385n board.

2.3.3 Protein Folding

The Protein Folding is one of the central issue in post-genomic era [29]. It is defined as the process of creating the tri-dimensional structure of the protein, starting from a sequence of amino acids. The goal of the prediction of the structure of a protein, through Ab Initio protein structure, is to identify the 3-dimensional structure of the protein basing only on its amino acid sequence.

Ab Initio protein structure means that the algorithm does not bases on pre-identified structures, as in the homology method, but it relies only on geometrical and energetic features.

This process is very important in biotechnology and drug discovery. However, although its importance, companies are slowed down by the high computation required by this algorithm [30].

The Protein Folding based on Ab Initio, the algorithm we are accelerating here, relies on Monte Carlo simulation (MCS). This methodology involves doing some random steps in conformation space. For each step, the degree of freedom of the molecule is perturbed and,

its energy changes. A step is hence accepted based on a probability that depends on the delta energy of the molecule.

The method of the algorithm we accelerated is based on an algorithm that compute self-collision in a deformable kinematic chain [31,32].

There are different ways of performing MCS of proteins. The differences usually are in these three aspects. The first one is the *Move set*. This is the type of changes that are applied for each step during the simulation. Then there is an *Acceptance Criterion* that is the rule the define which values are accepted and which are not. Finally, the *Energy Function* is what gives the score to the conformation of the molecule.

The version of the code presented in [33] uses two kinds of moves in the Move Set. The first one is the *Backbone Move* where the algorithm changes 3 angles at the same time. The degrees are chosen randomly from a zero-mean Gaussian distribution. The second move is the one of the *Rotamer*. Here a number of side-chains rotamers are changed at the same time.

Each step of the simulation performs a backbone move and some rotamer moves.

The Acceptance Criterion is the *Metropolis* criterion where:

$$k = \begin{cases} \Delta E \geq 0 & e^{\frac{-\Delta E}{kT}} \\ \Delta E < 0 & 1 \end{cases} \tag{2.5}$$

and, finally, the energy function is the one explained in [34].

The calculation of the energy of the molecule, that is the most compute-intense part of the algorithm (as will be seen later) happens in three stages.

The first step is the computation of the interaction forces of Van der Waals between the two molecules. The second step is the computation of the electrostatic energy between the atoms that are part of the amino acids. Finally, the last step is the computation of the solvation energy. This energy provides with information regarding how much a protein structure is stable in water[35].

2.3.3.1 State of The art

There are many different version of algorithms performing the protein folding. Many of these implementations, uses the homology method instead of the method we are using here. Some of this algorithms, perform an energy-based method that predicts the tertiary structure of proteins[36–38].

Our implementation of the algorithm is Ab Initio. In the state of the art, the acceleration of this type of algorithm is not so popular yet, however, we found two FPGA implementation based on Monte Carlo. Jain et al.[39] accelerated a protein energy minimization algorithm based on Monte Carlo called "Bhageerath". Their acceleration consists in a co-design of the algorithm on a Virtex 2 based PCI board. The function they decided to accelerate has been chosen after a profiling of the entire application and is the function that calculates the energy of the protein.

Given the initial configuration of the protein, the kernel calculates the potential energy for that configuration and proved back to the host a single energy value. The design has been made by decomposing the kernel in sub-units that have been implemented in three stages. Each stage has then been pipelined with the goal of achieving 1 result produced each clock cycle, and then everything has been linked together ensuring the synchronization of all the signals. Finally,

they exploited the fact that the calculation of the energy of each pairs is independent, hence can be done in parallel.

A second parallel implementation has been proposed by Zhu et al[40]. In the paper they parallelize an ab initio protein prediction algorithm called *BackboneDBN*. In this implementation they create an array of processing elements in order to compute the three phases of the algorithm in parallel.

The version of the algorithm they implemented, involves the computation of a matrix. This calculation, is the most intensive part of the whole program, hence this is the function they accelerate.

The acceleration, has been performed using a Virtex 5 - based accelerator. The acceleration strategy applied, is the parallelization of the three stages of the calculation of each element of the matrix. This stages are the dot product, a multiply operation and a divide one. Furthermore, they implemented an efficient way of hosting the columns of the matrix defined above in local memory, instead that in the block RAM of the FPGA.

CHAPTER 3

RELATED WORKS

In this chapter we provide an overview of existing tools used to guide the user to design an hardware application. Over the years, many tools have been created with the goal of alleviating the process of programming an FPGA. The majority of this tools perform a translation of a high level code, such as C or C++, and produces an optimized RTL implementation. Some of them, supports also system integration and runtime management.

We decided to compare some of the tool present in the state of the art, basing our analysis on some metrics. The metrics we decided are the following:

- **Commercial / Academic Tool**: an academic tool is usually open source, hence is usable by everyone without needing of licenses.
- **Interface Synthesis Support**: compilers with this feature, automatically create the interfaces that the implemented code will have to expose to communicate with external.
- **System Integration** and **Runtime Management** greatly alleviate the hardware design flow as they automatize its second step and manage how the program runs on the board.
- **GUI**: it can facilitate the coding of the algorithm and the introduction of optimizations to it.

- **C-Simulation Support** and **Co-Simulation Support** allow the user to check the semantic correctness of the high level code as well as the correctness of the generated RTL.
- **Pointer Support** allows pointer arithmetic and the possibility to write memory areas from different location in the code.
- **Synthesis based** on target platform: a code generated for a specific platform can be more performing with relation to a more general one but, at the same time, it is less portable.
- **Profiling** and **Debugging** Capabilities are key factors in identifying errors in the code, and understanding how to improve it further.
- **Arbitrary Precision** allow a reduction in the implemented architecture and a speed-up in the execution time.
- **Supported Language** is an important factor when the goal is to raise the level of abstraction of the hardware design flow.

As it is observable from Table II, the majority of the tools allow the user to input a high level code that is, usually C or C++. C and CO-Simulation, as well as profiling and debugging features, are supported by mostly all the tools. However, only one tool, among all the one considered in the table, perform system integration and runtime management. This tool is the Altera SDK. It allows a complete support to users in the hardware design flow, allowing the introduction of OpenCL code too.

TABLE II: COMPARISON BETWEEN TOOLS

Feature	ROCCC	Catapult C	Impulse C	Bambu	Vivado HLS	Altera SDK
Commercial / Academic	Acc.	Comm.	Comm.	Acc.	Comm.	Comm.
Interface Synthesis	✗	✓	✓	scripts provided	✓	✓
System Integration	✗	✗	✗	✗	✗	✓
Runtime Management	✗	✗	✗	✗	✗	✓
GUI	N.S.	✓	✓	N.S.	✓	???
C-Simulation	✗	✓	✓	✓	✓	✓
CO-Simulation	✗	✓	✓	✓	✓	✓
Pointer Support	N.S.	✓	N.S.	✓	✓	N.S.
Platform Dependent	✗	✗	✗	✗	✓	✓
Profiling	N.S.	✓	✓	N.S.	✓	✓
Debugging	N.S.	✓	✓	N.S.	✓	✓
Arbitrary Precision	N.S.	✓	N.S.	✓	✓	N.S.
Supported Language	C/C++, Fortran	C/C++	C	C	C/C++	C/C++, OpenCL

Now, we will present some of the tools we analyzed.

3.1 The ROCCC Framework

The Riverside Optimizing Configurable Computing Complier is a open source framework code generator that translates a subset of a C/C++ or Fortran code into RTL VHDL code than can be mapped directly to the FPGA fabric of a CSoC (Configurable System on Chip - a device usually composed of an FPGA and some microprocessors that can be programmed via software).

3.1.1 Framework Overview

The ROCCC framework[41,42] has been developed at University of California, Riverside. It is composed of a front-end that perform high level data flow analysis combined with loop transformations, and a back-end exploiting low-level parallelism and pipelining. It has two major objectives that are addressing the problem of writing code for the FPGA by hand and

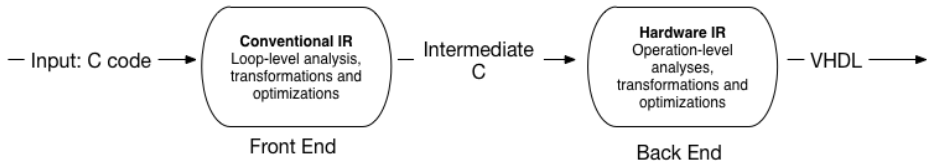


Figure 7: ROCCC 1.0 system overview

applying compile-time transformation to loops and multi-dimensional arrays that are in the code.

The tool limits to the generation of the RTL code, it does not, in fact, synthesize the produced code. The successive step of the hardware design flow must be performed with another tool.

A previous version of the framework, version 1.0, optimized the code for a specific target platform. However, this behavior revealed itself problematic as the complexity of the framework exploded with the increasing number of platform considered [43]. Hence all the information concerning a target platform has been removed.

The framework relies on the concepts of *modularity* and *reusability* and it is based on the LLVM (Low Level Virtual Machine) compiler. Its main goal is to maximize throughput, minimizing memory accesses and reducing final area usage. The way the framework produces the RTL is by means of *modules* and *systems*. Modules are hardware blocks with some defined input and outputs that performs a certain computation with a known delay. These hardware

blocks have the key factor of being reusable in order to build larger blocks. It is possible to directly integrate *cores* - hardware modules - that has been previously created by the user.

During the compilation flow, the compiler analyzes the code, identifying if there is reuse of data between different iteration of a loop. When this happens, the compiler allocates *smart buffers*, otherwise FIFOs are allocated.

Smart buffers are hardware elements that aims at reducing the memory bottleneck by organizing in a smart way the usage of data. In this structures, all the data that needs to be read more than once in the datapath is stored, so that there is not the needing to read from memory more than once.

As there is no information concerning a particular platform, the user has the possibility to add details himself. He has also the capability to choose the order of optimizations. In order to generate interfaces to memory the compiler has a *platform interface abstraction layer*.

3.1.2 Platform Interface Abstraction Layer

The framework can deal with two different interfaces: *streams* or *memory interfaces*. Streams are flow of data with the basic concept that there is no need of reading a value more than once. The second interface, is used when the program needs to access to non contiguous data, hence streams are not suitable anymore. For this second case, memory interfaces, paired with address generators, are created. The framework is able to synchronize the compute logic of the kernel, the datapath, with the reading from the interfaces too. It can, in fact, stall the computation until the data is ready. Other interfaces can be created, so that communication between different FPGAs can be performed. In this case, the user is asked to create some glue logic.

3.1.3 Modular Design and Hardware Optimizations

The bottom-up approach used by the framework allows to integrate modules to create a bigger system.

One of the important optimizations that the framework is able to perform is the *systolic array generation*. This is done by exploiting the flexibility and reusability of modules. The tool can automatize the majority of the steps in the creation of systolic arrays, facilitating the achieving of high performances. Another optimization is the *temporal common subexpression elimination*. It works by analyzing the code and identifying the parts of the code executed more than once. From this repeated codes, the compiler generates modules that are called more than once. This optimization reduces the amount of hardware generated by the compiler. Furthermore, it reduces the area occupied by the layout, affecting the final clock rate of the entire system. This reduction in the clock rate could result in a lowering of the throughput of the final system, hence this optimization is not performed automatically, and have to be activated by the user.[41,43]

3.2 Catapult C

Catapult C is an EDA (Electronic Design Automation) tool produced by the company Calypto Design System[44]. It automatically generates optimized RTL hardware descriptions for FPGAs and ASICs, starting from C/C++ code and aims at accelerating complex DSPs functionality[45].

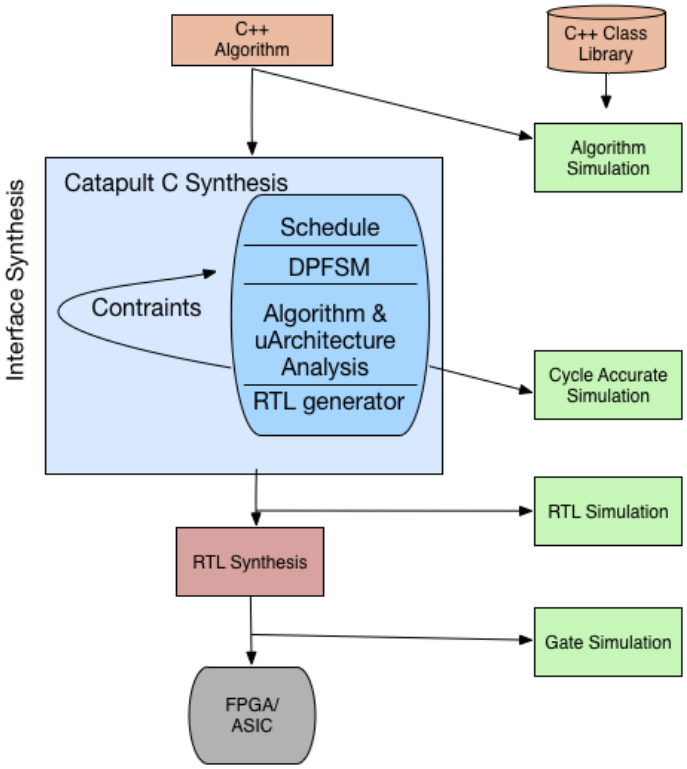


Figure 8: Catapult C design flow

The input code presents a main restriction. In fact it is not possible to use *dynamic memory allocation*. From the other hand, *pointers*, *classes*, *structs*, *array* and *bit accurate data type* are supported.

With this tool, the code is not targeted for a specific platform as the choice of the platform can occur in the future.

3.2.1 Design Flow

Catapult C design flow starts with an algorithm description using the ANSI C++ specification, the code can be untimed, no specification of the target platform, nor concurrency is required, and is necessary to describe only the functionality that it has to implement. All the hardware requirements, such as the communication interfaces and all the protocols do not have to be specified. Catapult C will in fact, apply all this kind of optimizations and specification during the synthesis process.

Once the algorithm has been described correctly the user is able to set some synthesis constraints. The first step is to set a target architecture to optimize the code for, set the target clock frequency, the clock enable behavior and information regarding the handshake protocol, at process level. During the second step is possible to set individual constraints to the input output interfaces of the design, loops, design resources and storage methodologies. During the step the user can explore the design space of the application by applying different kind of optimizations. All these enhancements can be set both from the GUI of the program or using a TCL script using the batch.

The tool exploit technology-specific libraries to create estimate of the resource and area usage of the target device. Thanks to this estimate, it is able to create a profiling of the application and a Gantt chart showing information about the architecture.

As Catapult C has a deep knowledge of the target architecture, it is able to implement targeted optimizations and to choose the best operators basing on the selected frequency of the system. It also provides the user with full control on the architecture that will be synthesized

into hardware, it is in fact possible to add different types of optimization such as loop unrolling or pipelining, array mapping, memory resizing (etc..) in order to test different micro-architectures so that is possible to get the maximum parallelization achievable.

The tool is technology aware, however the RTL netlist that is generated is not specific. As it is observable in Figure 8, Catapult C does not account for how the resources of the FPGA will be mapped and then used. Hence, it is on the ability of tools that performs the steps after the RTL synthesis, to optimize in a proper way FPGA resources.

One of the main limitations of this tools stands on the use of C++. In fact, the language does not contain operators such as the multiply-accumulate (MAC) or the multiply-add. Another one is that synthesis tools are able to infer only basic configuration of the DSPs. All the more advanced configuration for DSPs, are accessible by only hard coding the macros inside the RTL.

It is possible, however, to use Catapult C with the Altera Accelerated Libraries as explained in[44]. Thanks to this library, the designer has the possibility to target in a direct way DSP blocks inside the FPGA by simply using C++ functions.

Catapult C provides also a way to verify the RTL that is automatically generated, as well as the netlist created after the synthesis step. Furthermore, it is possible to create a System C infrastructure that test the original C++ design and the correctness of both the RTL and post-RTL netlist. The tool also gives the possibility to perform a co-simulation of the system C with the C++. The output of this simulation is compared with the C++ code that have been given to Catapult, to test the correctness of the implemented architecture. The flow generates

the SystemC code that will be used to connect and synchronize the signals as well as Makefiles that will compile all the scripts used to simulate the system.

3.2.2 Design Interface Synthesis

Catapult C automatically infers the type of communication interface to use. If the input port move data from the outside it can be a pointer to a read-only variable or a structure passed directly to the design. A similar thing will happen if the output port moves data to the outside of the design. If the input port can works as input-output it will be implemented as pointers that can be read or written. The task of Catapult C is to map the inputs of the C/C++ function to the hardware design that is implemented. In doing this it provides the user with the possibility to give some directives to the tools expressing preferences for the bandwidth, aspects related to the protocol used for the communication, such as the handshake, and the timing. The user has also control on mapping input/output arrays to wires, buses or stream or define some custom interface protocol[46].

3.2.3 Catapult LP

Catapult LP is the low power HLS solution of the Calypto company. It outputs a table view for all the power estimations of the design on the target board.

This tool born from the awareness that power optimization requires to consider all the steps of the hardware design, starting from the architecture to the implementation on the board. Usually, most decisions are made at the RTL level or below, however, it is not successful to follow this trend as this level is not enough for achieving optimal power savings. Catapult LP allow the user to search for the best low power architectural implementation at a higher level

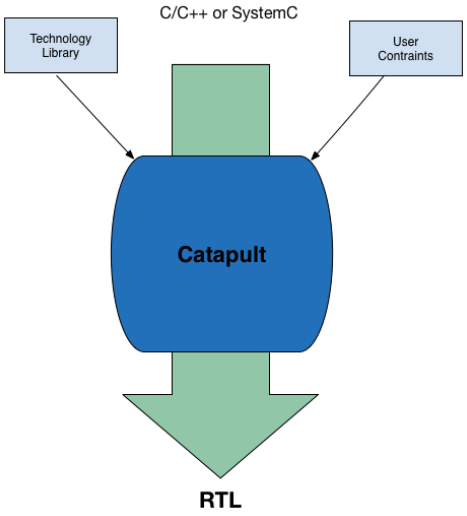


Figure 9: Catapult LP Synthesis flow

of abstraction and automatically implements optimizations as clock and memory gating at a lower level of abstraction.

3.3 Impulse C

Impulse C is a functional library, coupled with a compiler and some tools for debugging, produced by the company Impulse Accelerated Technologies. The tool is used to assist the development of highly parallel system that concern software and hardware parts. The tool accept as language ANSI C and include a compiler that takes care of translating the software C code into equivalent hardware description language. Processing units, called *processes*, are described using the C language, then the software connects them together to create a complete application that can be implemented on FPGAs. The tool can also create a software/hardware solution

that can be spread between software and hardware. Impulse C main focus is the mapping of algorithms on FPGA/processor systems that communicates by using streams, memories or signals and have a software side that resides in processors or desktop environments. It provides the possibility of performing simulation of the parallel application developed with the tool and it will be compiled by using the GCC compiler. It can target embedded processors as the Xilinx MicroBlaze, however it is not necessary to create an application that relies on an embedded processor. The tool generates code that run directly on a board too.

3.3.1 Impulse C Programming Model

Impulse C programming model involves an extension to the ANSI standard C, using a compatible library, paired with a defined model for the communication. The model is similar to the dataflow, and it creates an abstraction that facilitate the use of the tool to create the final algorithm. It emphasizes the use of data streams that are buffered. This streams are implemented as FIFO buffers and they allow the communication between different processes with a high level of abstraction. Although the design of this tool is oriented for stream-like application, it allows the usage of signals and shared memories to communicate. The communication model that the user will choose will be chosen based on the requirements of the application as the constraint of the target architecture.

All the code generated with Impulse C, can be compiled using standard development tools such as Visual Studio or GCC in order to have a software implementation. If the user wants to implement the code in hardware the user can use the compiler provided within the tool. This compiler, compile and optimize the code for a specific programmable hardware platform.

Figure 10: Impulse C Hardware Generation Flow

The tool provides also a debugging environment as well as tools for application profiling and co-simulation with other environments.

3.3.2 The Hardware Generation Flow

The hardware generation flow is showed in Figure 10 The first step of the flow is the *C preprocessing*. In this step, the code is analyzed, the macro expanded and all the references included. Then *C Analysis* is performed. The process that will be hardware and the one that will be software are identified as well as the communication interfaces, such as the streams, the signal and the shared memories. After this two steps there is the *initial optimization* step. During this step some basic optimization, such as dead code elimination, is performed only to the hardware processes. For what concern software processes they will be optimized later in the process using a different compiler. The *Loop Unrolling* phase, searches in the code for occurrences of the UNROLL pragma and tries to unroll the loop. If this reveals to be impossible, an error is generated. During the *secondary optimization* step, the tool performs a second optimization pass at block of statements level. Furthermore, the compiler checks for any automatic optimization between blocks. It may, in fact, collapse two different blocks into the same one for performance improvements. Finally, it checks for any occurrence of the

PIPELINE pragma and performs loop pipelining for the requested loops. The final step is the *Hardware Generation*; here the code is translated into HDL files. The files contain the description of the hardware processes and of the communication interfaces chosen. For what concern the information regarding the memory interfaces, they are referenced using hardware libraries provided for specific FPGAs.

Although all this steps facilitate the user into the design of a hardware/software, or completely hardware, application, the tool generates only synthesizable HDL. The user will have to manually import the code to another tool in order to perform the successive steps of the hardware design flow[47].

3.4 Bambu

Bambu is a semi-automatic framework, part of the pandA project[48,49] developed at Politecnico di Milano. It is used to assist the hardware designer in performing high level synthesis. The tool accepts the majority of the constructs defined in the C language and is equipped with the possibility to interface with some commercial tools for a synthesis that is aware of the technology. The framework is written in C++ and built in a modular way so that the user can implement different HLS algorithms. The user can also decide the order of performing the steps of the synthesis phase. In[50] it is said that the framework has the following advantages, relating to existing solutions. Firstly, the tool does not require the user to manipulate the semantic of the code written for the memory accesses; secondly, it is modular, hence highly customizable and extensible with other high level synthesis algorithms. Thirdly, it has a under

the roof method to check the correctness of the implemented RTL design. Finally, it is not bound to a target device it can in fact be configured to target different technologies.

3.4.1 The Bambu Framework

The architecture of the Bambu framework is depicted in Figure 11. It takes in input the source code, by means of C code, and an xml file where the user has to write all the configurations for the system he wants to implement. The output produced is the hardware circuit expressed in HDL and all the scripts necessary to perform the logic synthesis with the preferred third-party tool. The framework outputs also a test-bench, as well as the TCL script necessary to perform the RTL simulation. The framework can also generate the interfaces and the structures necessary for the integration of the accelerator with the rest of the system.

The front-end of the bambu framework interfaces with the GCC (GNU Compiler Collection) enabling the user to enable or disable different compiler optimization using the command line. Furthermore, the user can write the desired optimizations in the xml file provided in input at the framework. From the wrapping of the C code, performed using GCC, and the GIMPLE analysis, the call graph of the application is derived.

The second step consists in specific compiler-based optimization; firstly, a frontend analysis is performed for all the function that need to be synthesized. An example of this analysis is the lowering of operations necessary to compute memory addresses. This analysis is followed by the memory allocation phase.

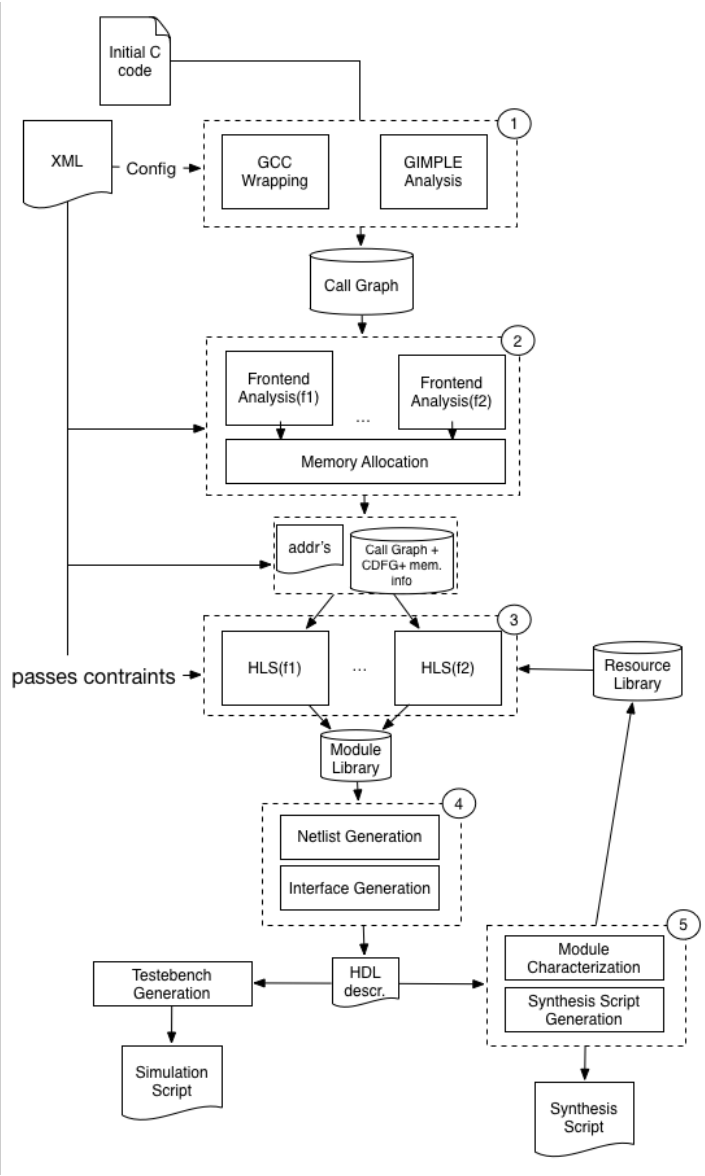


Figure 11: The Bambu Framework

The third phase is represented by the implementation of all the modules of the specification. For each of this module then, bambu produces the datapath, the finite state machine that works as controller and the memory interface.

All the modules that are produced in th third phase, are then combined together to create the final netlist and, on user demand, the interfaces that will be required for the system level integration.

The output architecture can then be integrated with many available software, both commercial and academical. In fact, in the fifth stage the tool integrates different containers that allow to use the developed architecture with commercial tools as *Altera Quartus* or *Xilinx ISE* (now Vivado). Basically, it can be integrated with all the software that accepts architectures based on a XML configuration file to generate custom feature in the logic synthesis. Bambu is able to generate the interfaces with some of the most famous protocol as *AMBA AXI* or *WISHBONE*. The final module is equipped with memory-mapped register used to write input and output parameters and various driver to communicate with the external.

The final step of the framework is the generation of a benchmark for the implemented circuit. This can be generated given the initial C configuration and a data set to test the code on. Hence the tool, after generating the RTL net, perform a RTL simulation and compare the result of this simulation with the result of using the same dataset on the software version of the code.

3.4.1.1 Supported Features

The bambu framework is able to support, in an efficient way, a good number of constructs that are usual to the C language. In particular, it is able to synthesize function calls and is able to reuse modules in case the function is called more than once. Memory addresses resolution is supported too, as well as pointer arithmetic. For what concern floating point arithmetic, both in single and double precision, is supported thanks to the integration of the FloPoCo Library[51]. Recursion is not supported, the only possibility to implement some recursion is in case the GCC compiler is able to transform it at compile time. Usually this happens only for simple recursions.

The hardware optimization that is possible to perform aims at reducing the area usage and increasing the overall performance of the system by exploiting resource characterization. Some of the optimizations that is possible to perform are pipelining, resource haring, operation chaining, etc... . The HLS flow can be even customized. The algorithm used and the flow of the synthesis can be customized entirely by the designer. After this steps, the user can check the correctness of the code and finally generates the implementation considering technology independent effects, such the one of the compiler, and technology dependent ones as resource characterization.

3.5 Vivado HLS

Vivado HLS is a high level synthesis produced by the company Xilinx Inc. It transforms a code specification, written in C/C++ or System C into a RTL (Register Transfer Level) representation that will be synthetized into a Xilinx FPGA. The aim of this tool is to improve

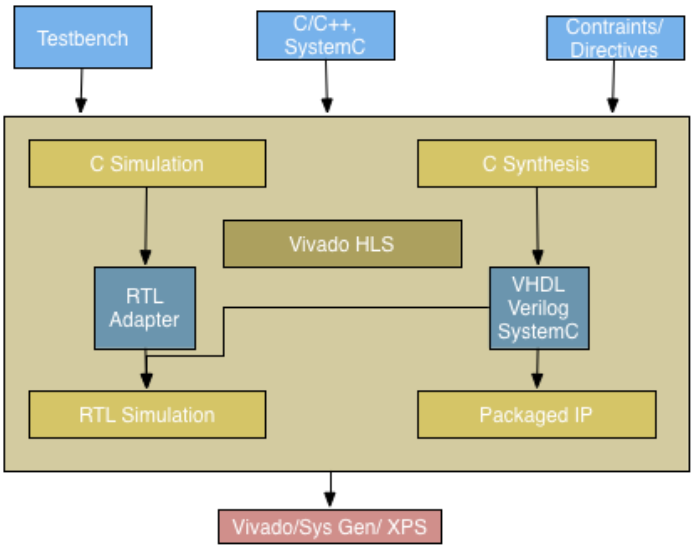


Figure 12: An Overview of Vivado HLS

the productivity of hardware designers and improve systems performance for software designers. In order to do so, it tries to abstract all the implementation details of the system, allows system verification at different levels, from the correctness of the input C-code to the final correctness of the RTL, and allows an easy exploration of the design space by allowing different implementation of the same code to be tested in parallel.

3.5.1 Features of the tool

The Vivado HLS tool produces an IP core that can be integrated into a hardware system and benefits from the integration with the Vivado Design Suite. It supports the design by providing the user with support on the language, as well as a set of feature that guide the user to create the optimal architecture.

The design flow of Vivado HLS starts with the choice a C algorithm, and then performs compilation, execution and debugging of it. Then the algorithm is synthesized into RTL. During this steps the tool creates reports and give the possibility to analyze the produced layout. The RTL implementation is then automatically verificated and finally, the RTL is packaged into an IP format chosen by the user.

3.5.1.1 Vivado HLS inputs and outputs

The tools need a C specification of the algorithm as input (Figure 12), as well as a file containing all the constraints for the system (target architecture, clock period and uncertainty, etc...). It is also possible to input a directives file; however, it is not mandatory to provide it. The directives file contains all the optimization chosen by the user, as well as the memory interfaces chosen. Finally, it is possible to provide test-bench files that will used for testing the correctness of the application.

All the input files specified above, can be given to the tool in two ways. They can be provided by using the Graphical User Interface, or using a TCL script in batch mode.

The main output of the tool is the RTL implementation of the system. This RTL implementation can then be synthesized into a gate-level representation and as a bitstream by performing logic synthesis. The files are packed into an IP block that can then be used with other Xilinx tools in order to generate the final bitstream to program the target FPGA board.

The tool generates also a System-C representation for simulation purpose. This simulation is cycle accurate and is called co-simulation. For the co-simulation, the tool compare the

generated System-C representation and check the final result with the test-bench that has been initially provided by the user.

After the synthesis, simulation and packaging steps the tool provides the user with report files.

3.5.1.2 Language and Libraries Support

The tool will synthesize the specified top-function and all the sub-functions that are in the hierarchy. As said before the language accepted by the tool is a C-description of the algorithm. However, not all the C constructs can be synthesized. Dynamic memory allocation is an example of construct that can't be implemented in hardware.

From the other hand, the tool provides with libraries that extend the standard C language. With Vivado HLS is possible to use Arbitrary Precision data type, there are function supporting video applications and mathematical operations and also support to Xilinx IP functions.

3.5.1.3 Synthesis Optimization and Analysis

The Integrated Design Environment (IDE) offered by Vivado HLS, allow the user to create projects that involve many solutions to the same algorithm.

It offers the possibility to apply many optimizations to the input code involving setting specific latencies for completion of loops, function and tasks; the specification of a fixed number of resources that are usable; specify if it is requested a pipelined execution of tasks, functions, instructions or loops and even remove the dependency of operations. All this optimization are specifiable directly using the Vivado IDE or can be written in a file that can then be given to the tool by using a TCL script.

The tool offer optimization also at I/O level. The user can pick among different communication protocols and can directly specify the interface that the tool will implement.

After specifying all the different optimizations, the code goes through the synthesis process and the tool produces automatically reports to provide the user with information concerning how the optimizations affected the system. The tool provides a graphical tool in order to interactively analyze in detail the results.

3.5.1.4 RTL Correctness Verification and System Package

Once the RTL is generated, the user would want to check its correctness. The tool gives this possibility. If the user gave as input also a test-bench, the tool will generate System-C code in order to compare its result to the rest given by the test-bench. In case there is mismatch in the results, the tool returns an error. There are many simulators available within Vivado HLS and the user can choose among one (or more). Once checked the final correctness of the generated RTL, the user can use the export features in order to create an IP package. Vivado HLS gives the possibility to package the IP in 5 different formats, supported by different tools provided by Xilinx. For some of the formats, it is possible to perform *logic synthesis*. This step ensures that timing and area estimates, generated from Vivado HLS, are correct, and hence that the IP will not have any problem in the next steps.

3.6 Altera SDK for OpenCL FPGA Programming Flow

The Altera SDK for OpenCL is a software produced by the company Altera. It abstracts away the traditional hardware flow for FPGAs automatizing the design process and decreasing the development time, allowing a software emulation of the source code on the host computer

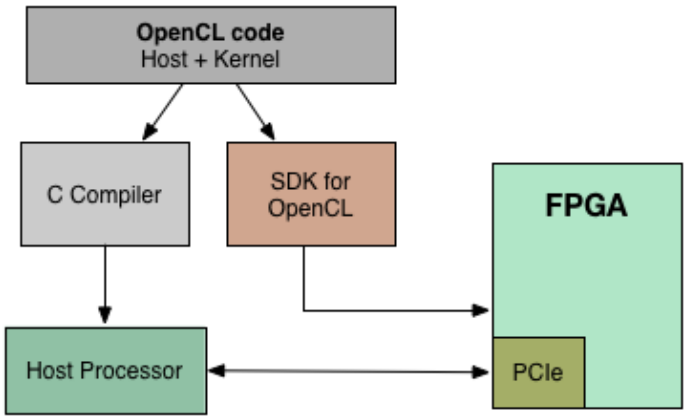


Figure 13: Altera SDK for Opencl Architecture

and leveraging the optimized OpenCL API to implement the final kernel. The general flow of the tool is showed in Figure 14.

Given an OpenCL code, the tool performs the steps oh high level synthesis and system level design. It is used to program Altera FPGAs with OpenCL applications. The tool works as a two step process. In the first step, the *Altera Offline Compiler* takes the OpenCL application, called kernel, and compiles it. Secondly, on the host side a C compiler takes the host application, used to control the kernel, and finally links the compiled OpenCL to the compiled C host code. The host application is the one that, exploiting the OpenCL APIs, will take care of the runtime management. Figure 13 shows an overview of the architecture used by the tool. The host computer and the accelerator communicates over PCIe. All the inputs and output of the kernel, that have to be passed to the host computer pass over this cable. The kernel then will be implemented on the FPGA where the real acceleration takes place.

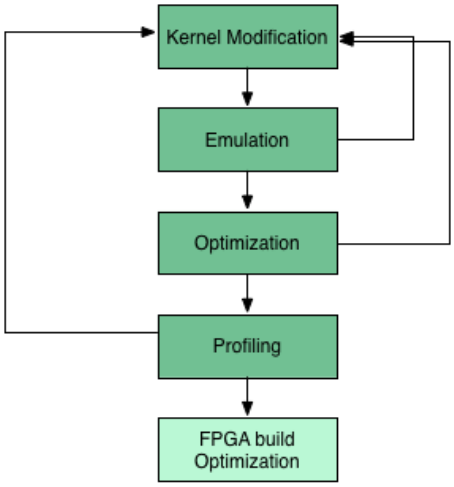


Figure 14: Altera SDK for OpenCL general design flow

3.6.1 The Compilation Flow

The Altera Offline Compiler is able to create the hardware configuration for the target FPGA in a *one step* process, or in a *multi-step* one. It depends on the complexity of the kernel code, and the type of implementation the user wants to perform.

The one step process is used when it is not necessary to massively optimize the kernel code. Given the code, the compiler creates the hardware configuration in just one step. The operation performed by the compiler are the one in Figure 15. In this case the compilation process takes hours and in case of syntactic errors it needs to be started from scratch. It also generates report regarding the estimate resource usage and, in case a single OpenCL work-item is used, the tool identifies the bottlenecks of the system.

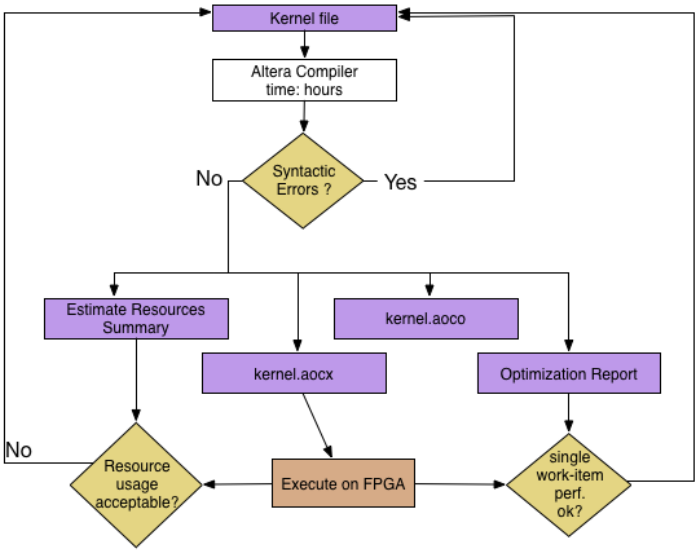


Figure 15: Compilation Flow for the one step process

In case multiple iteration are needed to implement performance optimization, it is suggested to perform the multi-step process.

Figure 16 shows the steps of this second process. It is observable as the flow is broken into intermediate steps, allowing intermediate evaluation of the kernel in order to maximize the optimizations without performing a complete compilation each time. This process is broken into 5 steps: *Intermediate Compilation*, *Emulation*, *Profiling* and *Full deployment*. The intermediate compilation is limited to a syntactical correctness of the code. During this phase is provided also an estimated resource usage summary. The second phase emulates the OpenCL code on multiple emulation devices on the x86-64 host. The profiling phase instruments performance counters into the generated Verilog code and, finally, the final phase performs a full compilation,

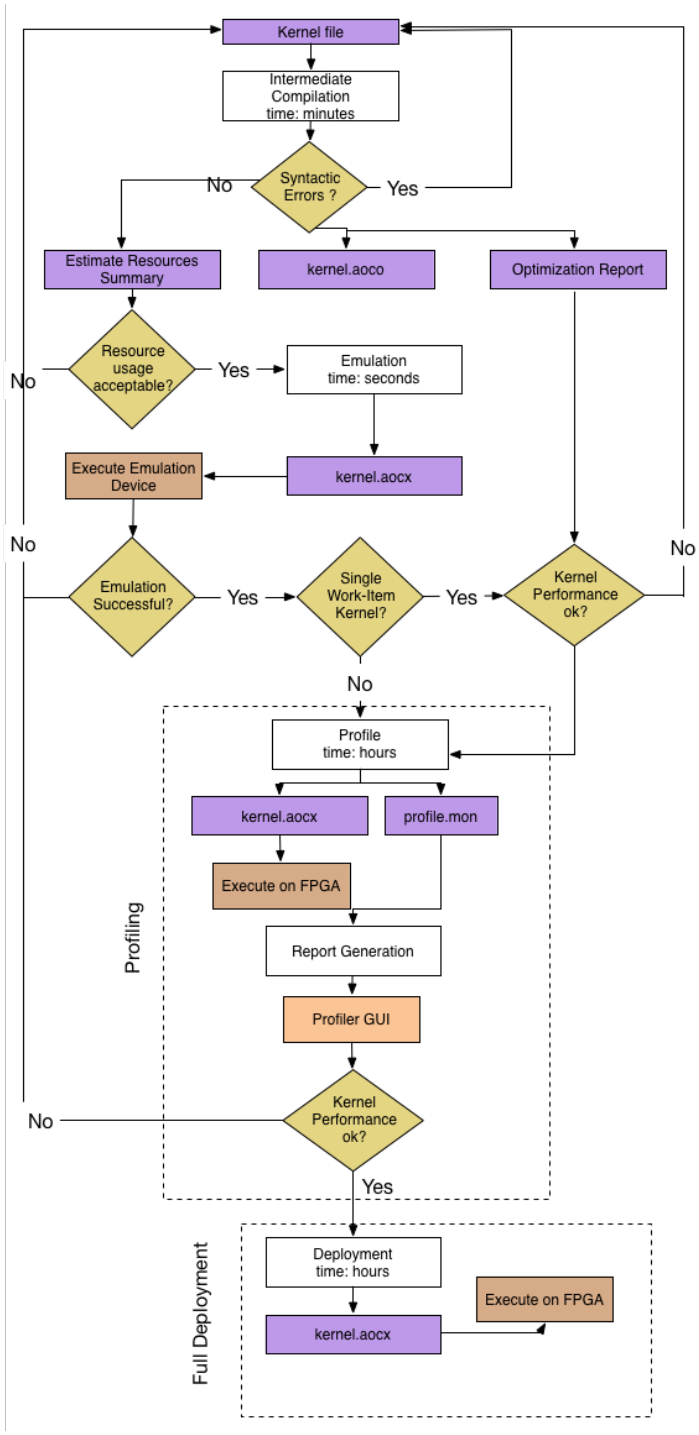


Figure 16: Compilation Flow for the multi-step process

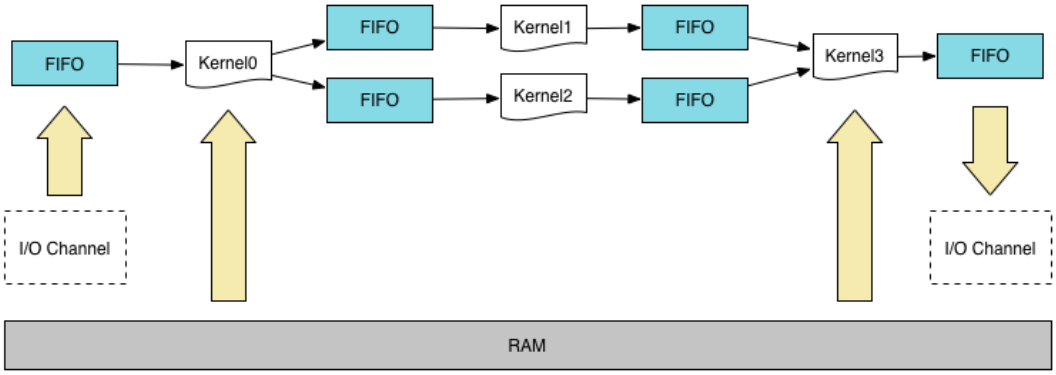


Figure 17: Example of the Altera Channels

giving, as final product, a *.aocx* file that can be directly executed on the FPGA. During each of the phases explained above, if the user is not satisfied with the performances of the kernel, he can step back into a previous phase of the development in order to optimize further the kernel.

3.6.2 Optimizations

Multiple optimizations, targeting memory access and data processing are doable. The compiler is able to apply optimizations like loop unrolling, work-group size and compute unit number tuning in order to increase the computational resources that can be used on the FPGA board. For what concern memory optimizations, it can operate so that the efficiency in the memory access is increased by specifying the size of the pointer located in local memory or by specifying the location of the buffers in global memory.

The second optimization regards on which type of memory, DDR, QDR the data have to be placed. Choosing a type of memory over another can considerably affect performance.

A considerable extension to the OpenCL framework introduced by Altera is the *AOCL Channel Extension*. This extension allow communication between different kernels on the same FPGA board decoupling the kernels execution from the host computer.

A noticeable difference introduced by the channels is that the host computer does not need to control and synchronize the data movement between kernels. An example of the channels can be seen in Figure 17. They are implemented as FIFO buffers that holds the data on the FPGA device so that it can be accessed among different work-groups and ND-range invocations. However, the data is not persistent among different invocation of the kernel. As the channels are implemented as FIFO buffers, they follow the producer-consumer paradigm. The user will have to avoid deadlocks while using this extension by adding memory fences or barriers in order to enforce an order on reads and writes.

Some problem may rise in the order of execution when using the Altera channel, hence it is possible to introduce a order, called the *work-item serial execution*. This is a sequential behavior that order the work-items based on the sequential ID in the compute unit. In case the channel is implemented inside a loop, the compiler ensures that all loop iterations i of each work-item in a work-group executes before iteration $i + 1$ of each work-item in a work-group.

Obviously some restrictions apply in the implementation of channel extensions. The first restriction is that there need to be a *single call site*.

Secondly, if a channel is implemented into a loop, the cycle can not be unrolled. Channels can then be read or write only.

Static indexing is required. It is not possible to apply dynamic indexing to arrays of channels IDs, hence the id of a channel need to be statically defined. Kernels that use channels can not be *vectorized*.

Altera channels are an extension of the OpenCL standard, hence the user has to explicitly express the intent to use them. They are blocking buffers, and is possible to determine the *depth* of the channel. The depth is the maximum number of elements that can be stored in the buffer before it stalls the execution. The user can choose to implement the channels as *non-blocking buffers* too.

The channels can also be used to interface with input/output features of an FPGA board. Examples of this features can involve PCIe, cameras or even network interfaces.

The big advantage introduced by the Altera Channels extension is the following. The update in the global memories, are not guaranteed to be consistent until the kernel terminates its execution. This means that there is no guaranteed that two kernel that execute in parallel, sharing data in the global memory will produce correct results. In this context, channels can be used to share the data and to guarantee that the data that is moved from one kernel to the other one is consistent. In fact, by using the channels, the kernel will not have to write data in global memory.[22,52]

CHAPTER 4

PROBLEM DEFINITION

In this chapter, we analyze the problem presented in this thesis. The first section starts by talking of the design for HPC while the second one, talks about how new CAD tools aims at improving productivity by performing the majority of the steps concerning the hardware design flow.

4.1 Design for HPC

Nowadays trend to accelerate High Performance Computing applications is to pair a CPU with one, or many, accelerators. The CPU and the accelerator split the computation of an application so that all the compute intense parts of the algorithm are processed on the accelerator, while all the control functions and the part of the algorithm that does not need to be parallelized are placed on the CPU. In this context, the accelerators that can be used to parallelize algorithms are many. Graphical Processing Units (GPUs), co-processors as the Intel Xeon PHI and FPGAs are some examples. Some of the reasons that may influence the choice of one platform over the other are performance, power consumption and programmability. Usually, the user will choose what it will be the best trade-off among these three features.

Figure 18 shows an example of performance/watt comparison between CPU, GPU and FPGA, the data regards a benchmarking performed by Yihua Fang and Xiao Bo Zhao for a university project concerning the creation of a heterogeneous architecture [53]. The kernel

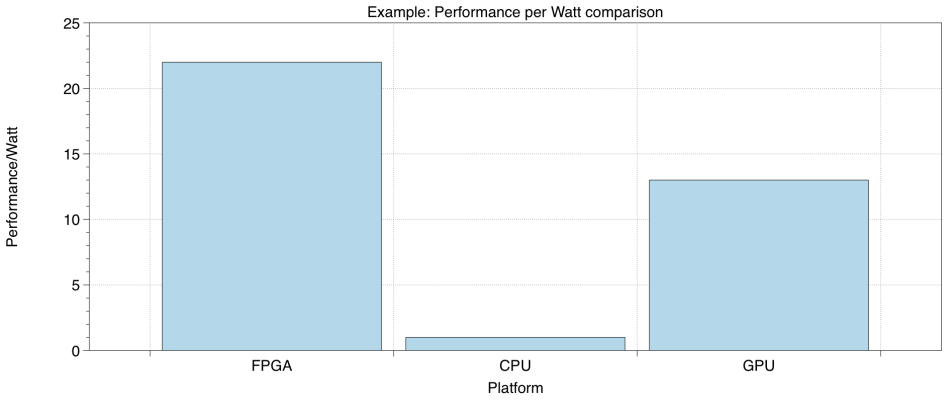


Figure 18: Performance/Watt comparison between platforms

implemented in their project involves a core that cracks the SHA1 hashcode. The data regarding the CPU has been benchmarked on a Intel Xeon CPU with 6 cores running at 3.2 GHz, the one concerning the GPU has been taken using a RADEON 7970, while the data of the FPGA has been benchmarked using a cluster of Altera DE2-115 Cyclone IV.

In the chart, it is clear that the highest ratio of performance over power consumption is achieved by the FPGA, followed by the GPU. The lowest value is the one of the CPU. The FPGA can leverage its reconfigurable architecture and power efficiency to accelerate applications as in the field of HPC. GPUs have high parallelization capabilities, but their power consumption is considerably higher compared to FPGAs. However, as it is understandable from the information provided on *The Green 500* [54], the most energy-efficient platforms are composed of GPU, CPUs and co-processors, no FPGA is used. The Green 500 provides a ranking of the most energy-efficient supercomputers that exist in the world. Its goal is to provide a list that relies on factor

that does not only involve performance. Instead, it takes under consideration parameters as performance/power consumption, reliability and energy efficiency of the supercomputer.

Nevertheless, FPGAs revealed to be a valid support to the computation in particular applications, both in academical and industrial scenarios. Although they do not appear in The Green 500, it has been demonstrated that they suite particularly well in field as High Performance Computing[55].

Someone may ask why, as the FPGAs are clearly the best platform in terms of performance over power consumption, they does not appear in the list of the most power-efficient supercomputer in the world. The reason resides in the fact that FPGAs are platforms that are hard to program. Their design flow involves the translation of code into HDL code and the consequent system level design step. This second step involves the implementation of the code into a specific FPGA. This flow is complex, time consuming and error-prone. However, a tool that completely automatize this process, raises the level of optimizations that can be performed, and this may result in a lowering of the achievable performance in some cases. Companies, such as Xilinx, Altera and Maxeler, produced design suites to alleviate this task and automate the design of application for FPGAs, starting from the translation of the code, till the integration on the FPGA.

4.2 System Level Design and Integration

Figure 19 shows the classical design flow for FPGAs devices. The first steps, involving the translation and optimization of a C/C++ code into HDL in order to create IP cores, is part of the high level synthesis. Then, the generated hardware core will need to be integrated into

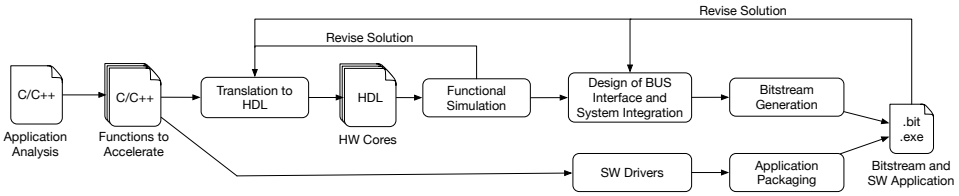


Figure 19: Classical Design Flow for FPGAs

the FPGA board by creating the surrounding architecture and memory infrastructure. This second phase is called System Level Design and Integration.

System Integration (or System Level Design) is the second step of the design flow for FPGAs. Once that the IP Core (or Hardware Core) has been generated from a high level synthesis tool, the user will still have to manually perform the integration step and he will have to manually handle the runtime, write the software driver and generate the bitstream. As opposite, when designing applications for GPUs, the user has a programming environment that completely abstracts all the phases needed to program the device. In this context, the user has only to write the code that concerns the algorithm that he is accelerating. The tool then takes care of all the compilation and integration steps. Usually, applications of this type involve the use of high level code as the one provided with Nvidia CUDA [56] or the OpenCL project [57]. Using these languages, the user can address all the problems that concern the runtime management of the application. It is clear that this tools highly facilitate the design of application for GPUs. For this reason, the new trend is to create a new environment for the design flow on FPGA that is the more similar as possible to the one used for GPUs. The goal is to create a tool that

alleviates all the problems concerning system level design, and then the final integration of the design system on FPGAs. The tool has to take care of the majority of the design step shown in Figure 19 so that the user can focus only on optimizations at code-level. Furthermore, the tool has to exploit a high level, portable code such as OpenCL so that FPGAs can be addressed to a much wider audience. Programmers used to the flow introduced for GPUs, would then be able to program FPGAs, with just a little knowledge of the hardware architecture they will have to program.

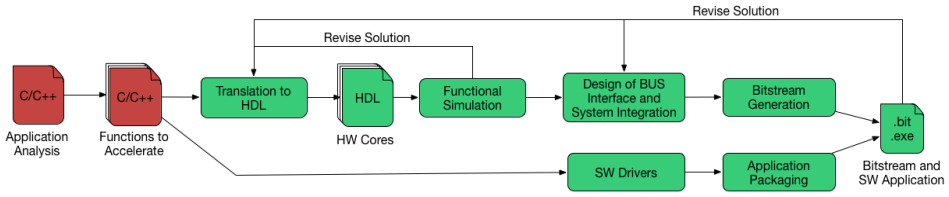


Figure 20: Xilinx/Altera Design Flow for FPGAs

Vendor companies, such as Altera and Xilinx, already addressed this problem and developed softwares that guide the user in the design process targeting FPGAs. Their tools allow a high level specification of an algorithm and automatically performs many of the classical design steps.

An example of the design flow introduced by these new tools can be seen in Figure 20. In read are highlighted all the operations that need to be performed by the user, while in green

the operations done by the tool. The reader may notice the reduction in effort that the user has to perform to program the FPGA device.

Both the tools presented by Altera and Xilinx, aim at alleviating the burden of FPGA programming, however, for this work, we decided to use SDAccel, the software defined accelerator produced by Xilinx. The reasons that leaded us to this decision are that SDAccel provides with a complete support, that starts with the introduction of the high level code and end with the runtime management on the FPGA board. The tool automatically perform system level integration and runtime management, alleviating to the user the programming of the FPGA. Furthermore, by using SDAccel we are not bound to a programming language. We can, in fact code our algorithm in OpenCL, or using C and C++.

Although SDAccel is a valid tool for designing and accelerating application on FPGAs, it has not been straightforward to develop applications with the tool. There have been many challenges that we had to face during the development phase. First of all, we have to note that SDAccel is a software that has not been released yet, hence many functionalities are not completely working. The OpenCL version implemented in the release of the software we used, is the 1.0. Hence, it has not been possible to exploit the full potentiality of the tool as many communication structures have not been implemented yet. Furthermore, many of the implemented functionalities are still under development, limiting considerably the expression capabilities of the designer. Due to problems to the communication interfaces implemented in the SDAccel framework between the host and the accelerator, achieving the full bandwidth is challenging. Many of the optimizations that were available with the Vivado HLS tools, were not

applicable with OpenCL, or were still under development and finding turnarounds to problems is a slow process. Furthermore, the way to solve this issues bring to a coding style that is non-intuitive and require more development time.

CHAPTER 5

AN SDACCEL-BASED DESIGN FLOW

In this chapter we will give an overview of the flow introduced by SDAccel. We will talk about our contribution to the development of the tool and explain how this tool has been used in our work. Finally, we will talk about how we faced the problem explained in the previous chapter.

5.1 SDAccel

Xilinx SDaccel is the new tool developed by Xilinx that provides a new and innovative design flow targeting FPGA devices (Figure 21). It is a Software-Defined Development Environment that targets FPGAs starting either from a C, C++, System C or OpenCL language. The software side of this tool is based on the OpenCL runtime and APIs; this makes SDAccel a development environment for FPGA which totally resembles the classic design flow to target other accelerators such as GPUs. The SDAccel environment comes with a Eclipse-based GUI and a fast compiler that is optimized to efficiently use the area of the FPGA. [58]

SDAccel leverages on the *Vivado Design Suite* tool from Xilinx, exploiting Vivado HLS tool for the High Level Synthesis step; Vivado HLS, which is already capable of handling C and C++, has been extended with the capability of supporting the compilation of OpenCL code. The extension of the compiler allows the user to implement any native OpenCL compiler optimization such as loop merging, loop flattening and unrolling, as well as more advanced op-

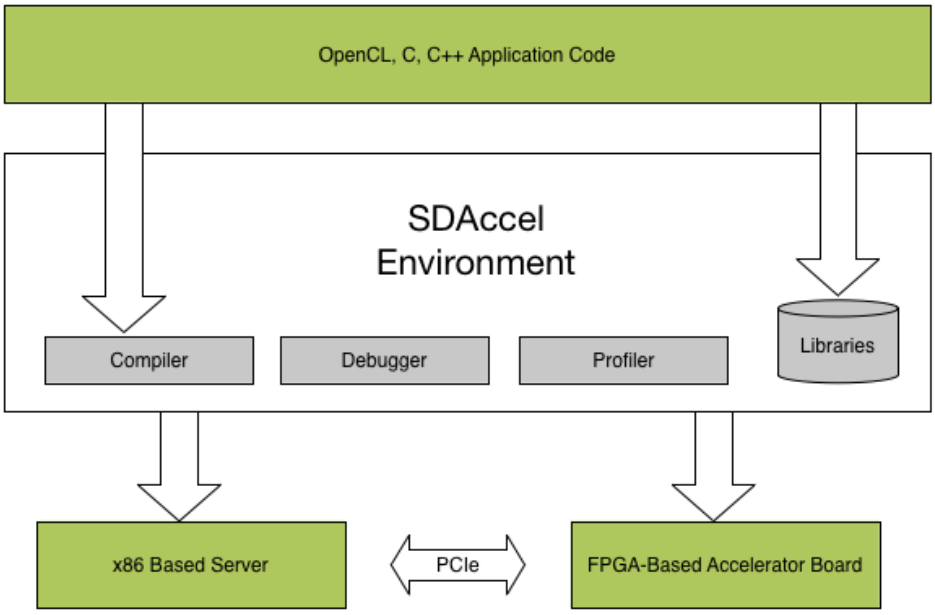


Figure 21: SDAccel software environment overview.

timization regarding memory usage, loop pipelining and dataflow optimizations. The introduction of OpenCL source codes allows an easier programming environment to all the programmers who are familiar with OpenCL and used to the memory hierarchy abstraction introduced by this framework. At the same time, it allows full access to FPGAs features and exploits the portability of OpenCL. It is, in fact, possible to write the code once and target different devices as GPUs and CPUs.

The final architecture produced by SDAccel is fully compliant with the OpenCL standard, and it is composed as shown in Figure 22. In particular, SDAccel presents a fundamental difference with respect to the standard OpenCL programming environment. In fact, standard

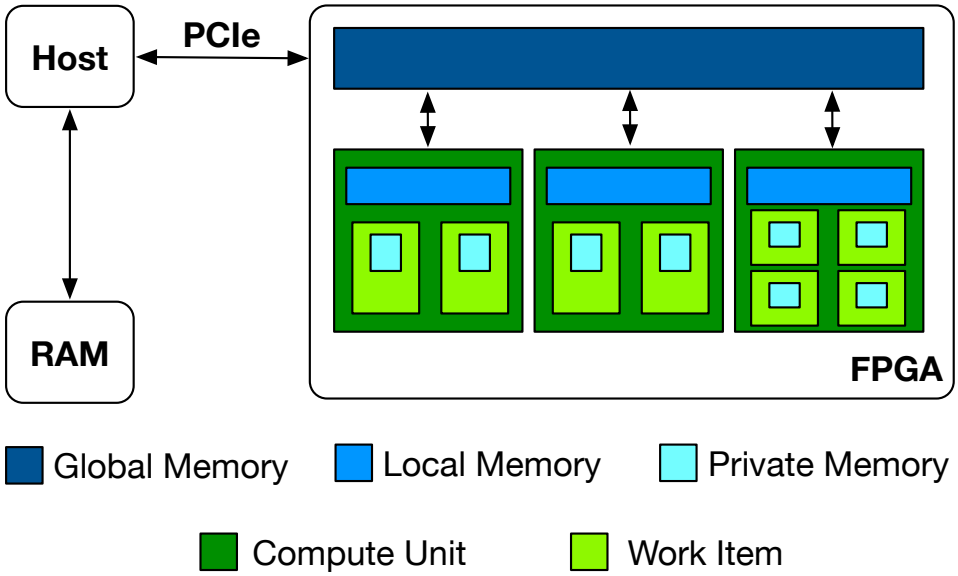


Figure 22: SDAccel architecture and memory hierarchy

OpenCL relies on target with a fixed hardware architecture while SDAccel, targeting FPGAs, can exploit their non fixed architecture by customizing and reprogramming them.

5.1.1 OpenCL Platform Model

The concept of platform is a logical abstraction of the hardware executing the OpenCL code. It is defined as a group of elements composed of a processor and one or more devices. In the OpenCL abstraction, the *host* processor is the element where the operating systems runs on and where the task are launched. The OpenCL kernel are then usually executed on a *device*. The device is then divided in *compute units*. A compute unit is the element where a *work-group*, element composed of *work-items*, is executed. It is also divided in *processing elements*.

Each processing element is the element responsible of executing all the operations concerning a work-item.

In the Xilinx SDAccel environment, the host processor is an x86 machine communicating with the device, an FPGA, through the PCIe. It is responsible for enabling the driver for the devices, managing the OS, executing the host code, creating the memory buffers and controlling the data transfer to and from the host code. OpenCL provides a *runtime library* that has to be linked at compile time, hence the user can perform the operations described above using APIs. This means that the APIs are portable across vendors but the implementation of the functions, that is provided by the vendor of the used platform, may be different.

FPGAs, are not limited by the architectural constraints of CPUs and GPUs. However, in order to support the OpenCL standard they need to have the following characteristics. Firstly, there is the needing to have a connection to the host processor, hence I/O peripheral are required. A kernel region, defined as the area where the user-defined kernel will be implemented, need to be identified and all the interconnections and the memory controllers need to be provided. SDAccel has some defined devices available. This devices are based on Virtex-7 and Kintex-7 FPGAs. Furthermore, there is the possibility to create new FPGA devices or add third-party created devices.

5.1.2 SDAccel - OpenCL Memory Architecture

In Figure 22, it is possible to identify all the memory related blocks in blue. It is observable, that they respect the OpenCL memory abstraction. In fact, three different layers of memory are identifiable: *Global Memory*, *Local Memory* and *Private Memory*. The host and the accelerator

are connected over PCIe (Peripheral Component Interconnect Express). The global memory is located on the device and is accessible by both the accelerator and the host. The other two layers of memory are accessible only from the device. There is a local memory available for each compute unit and, internally to the compute unit, there are different work-items. Each work-item has a private memory that is available only for itself. There are other two layers of memory. The *Host Memory*, that is the memory of the host platform, and the *Constant Memory*. This second memory is a particular type of global memory that can not be written from the device. The FPGA resources are divided among the so called *compute units*. Each compute unit implements the code to execute one kernel, however the same kernel can be implemented by different compute units. This one-to-many relation translate into the parallel execution of the kernel code on all the compute units, where the code is implemented, in parallel. Furthermore, each compute unit is composed of a variable number of *work-items*, indicated in light green in Figure 22. The user has the possibility to customize the number of work-items in the compute units. The work-items are the element that are effectively carrying out the kernel execution. They execute in parallel inside the compute unit. [59,60]

5.1.3 SDAccel Compilation FLow

The main advantaged of using SDAccel is that the user can be agnostic of FPGAs in compiling OpenCL application. However, in order to maximize the performance, the designer should have a minimum understanding of the accelerator. As SDAccel manages all the low level details in the compilation of the program, the user is not requested to have any further knowledge beyond what is expected for the same compilation using a CPU or GPUs. SDAccel

can be used by the provided GUI, or the commands can be given from a terminal by giving a TCL script as input. The steps to follow to compile an application are the following. First a *solution* is created and a *device* is defined. Then the user needs to specify all the files that represent the OpenCL host code. Here a file can be a C/C++ code and its headers. Hence the user need to define the files that represent the kernel. As said above, a kernel can be written in C/C++, System C or OpenCL. If the user defines a C/C++ kernel, he will have to define also the memory interfaces by using dedicated pragma provided by SDAccel. The pragmas are the same usable in Vivado HLS.

Once added the files, the kernel has to be added to the solution previously created. The kernel must be wrapped in a container by creating the *Xilinx OpenCL Compute Unit Binary Container*. Here there is a small difference between OpenCL kernel compiled for CPU/GPUs and the ones compiled for FPGAs. CPUs and GPUs relies on a fixed architecture, hence the compilation of the kernel code can happen online, directly from the source code. From the other hand, the FPGA is a white canvas, furthermore the compilation for such device would take too much time to be performed online. Many optimizations could be exploited and the time for having a compiled architecture would be too much. For these reasons, OpenCL addresses this problem by providing a way to compile the kernel offline, generating a binary file that can be then given as input to the host code. Xilinx has defined the *Xilinx OpenCL Compute Unit Binary Format* to identify this binary files. Its extension is *.xclbin* and it has to be compiled offline with a provided command in the TCL script.

Once the kernel has been defined and the command to compile it given, the user will have to choose the area, on the FPGA, where the kernel will be placed. This is done pairing a region with a .xclbin file, and then creating a compute unit for the specific binary.

After all these operations have been performed, the system can be built, packaged for the FPGA, and finally run.

Before building the system, the environment introduced by SDAccel allows two level of simulation of the system: the *CPU Simulation* and the *Hardware Emulation*. The first one test the functional correctness of the code. However, is possible that a code whose CPU simulation succeeds, will not pass the hardware emulation. This second emulation is an RTL simulation. It tests the correctness of the code on the FPGA board by simulating the hardware.

For both this simulations, is up to the user to check the correctness, in terms of results, of the code.

5.1.4 Optimizations

The OpenCL kernel guarantees code portability but not performance portability. This means that the code has to be optimized when ported to another platform. When trying to optimize the kernel for a CPU/GPU, the programmer will have to optimize the kernel for a fixed architecture, on the FPGA the story is different. The designer will have to take into account that there is no fixed architecture on an FPGA.

With SDAccel is possible to optimize the system at different layers. The two main layers of optimization are the kernel and the memory architecture.

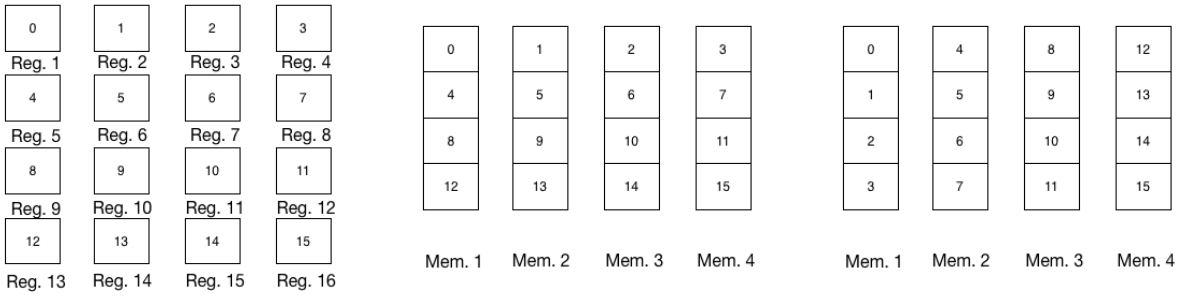


Figure 23: Example of partitioning

For what concern the kernel the first optimization that is doable is the *tuning of the work-group size*. This optimization defines the size of the ND range space of the kernel, and can be tuned with only one command. When the command is given, Vivado HLS, that is under the roof of SDAccel, start transforming the code by creating three loops to handle the three dimensional space created by OpenCL. It is possible to have either variable of fixed loop bound. The difference is that by having fixed loop bound, SDAccel can optimize size of the local memories in the compute unit and provide precise estimate of the latency in the report.

Loop Unrolling and *Loop Pipelining* are doable too. It also possible to perform *work-item pipelining* that is an extension of the loop pipelining that acts on the kernel work group. The optimization works by pipelining the instruction of the work-items in a work-group and the execution of different work-groups.

A graphical example of what is a loop pipelining is visible in Figure 24. The first one is how the execution works before applying the pipelining optimization while, the second picture

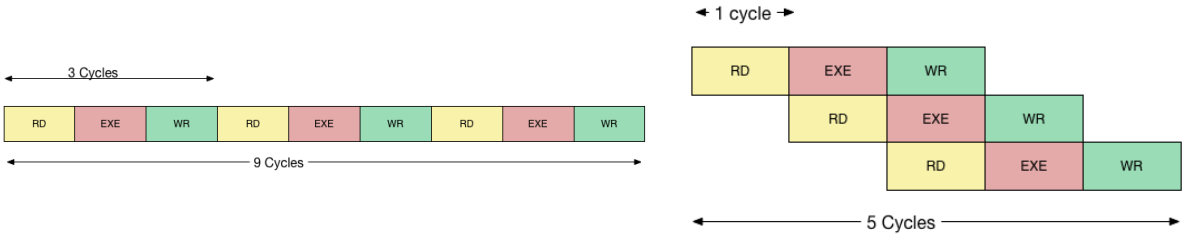


Figure 24: Example of pipelining

shows how it works after the optimization. It is noticeable how the total number of clock cycles is reduced.

Barriers are another OpenCL construct that is implemented in SDAccel. They are needed to ensure the consistency of the memory. In OpenCL, there is no order of execution as all the execution is out of order. Hence, there is the needing of a construct that works as a check point for all the work items in a work group. In SDAccel, this kind of optimization is performed by simply executing the loop nest in a sequential manner.

SDAccel takes care also of defining all the memory interfaces as well as the connections of the memory interfaces with the logic of the FPGA. Also at this layer, it is possible to optimize the hardware architecture. The main optimization doable at this level are three: *Multiple Memory Ports per Kernel*, *Adjustable Bitwidth of Memory Port* and exploiting *On-Chip Global Memories*.

The first optimization acts by increasing the number of physical link of the input output ports. Usually, SDAccel generates the best layout by using the least amount of area of the

FPGA. However, if this behavior decreases the amount of area usage, it can also decrease the performance. Hence, this optimization aims at increasing the performance by augmenting the parallelism. It is not performed automatically as it requires knowledge, from the designer, of the target architecture.

SDAccel offers the possibility to change the size, in terms of bits, of the memory port that will be implemented. The default behavior is that when the function is:

```
void function ( __global float16 * input, __global float16 * output)
```

SDaccel set the bitwidth of the ports, to the native type of the input/outputs. In this particular case, the native type would be float, hence SDAccel would set the bithwidth to 32, and would require 16 memory transaction to read the entire *float16* element. The user can decide to augment the bitwidth to values as 32,64,128,256 and 512 bits. In this particular case, by setting the interface to 512 bit it would require only a memory transaction to read one element from the input.

The user can define some global memory, used to send data between different kernels. In case it is not necessary that these memories are visible from the host platform, SDAccel automatically optimize this On-Chip Global Memories by moving them from the DDR memory to the logic of the FPGA, considerably increasing the speed of moving data.

Targeting an FPGA, rather than a GPU, present advantages also in the customizability of the memory architecture at different levels, from the system, down to the compute unit. At this level in fact, it is possible to apply some *partitioning* of the memories. Partitioning the memory means that SDAccel will create multiple physical array rather than an entire block of memory.

The tool allows three levels of partitioning, namely: *cyclic*, *block* and *complete*. A complete partitioning (Figure 23 - first picture) will create independent registers for each element in the array while a block partitioning (Figure 23 - second picture) will create N physical memories composed of M elements each, with M number of elements in each memory. Finally choosing a cyclic partitioning, observable in Figure 23 - last picture, tells SDAccel to spread the element of the array over N physical memories composed of M elements each. Here M is defined as in the case of a block partitioning. The difference is that the elements are not place sequentially, as in the block partitioning, but they are placed following the *card deadline* paradigm. The advantage of performing a partitioning of the memory is that, in this way, the architecture will be able to access more than one element at the same time, without the limitation of the BRAM. This will result in an increase of the parallelism of the system. [60]

5.2 Problem Solving and Contributions

The development using the SDAccel framework had been challenging. The tool has not been released yet, hence the software presents some bugs that limit its capabilities. Furthermore, the support that a designer could find on the internet for certain type of problem is missing, as not many people has access to this framework. Here we will present some of the solution that we applied to face the problems introduced in Chapter 4.

In order to face the limitation of the OpenCL language introduced by Xilinx in the SDAccel framework we decided to code our kernel code in C/C++. However, the host code of the program is coded in OpenCL and exploit the OpenCL runtime APIs. The change of language

revealed itself necessary because many of the OpenCL functions provided where not working properly and were not event optimized to achieve the best performances.

Another problem in using OpenCL was that many of the optimization available for the C/C++ language, were not available to use in OpenCL.

The framework presented a problem with the communication interfaces too. Firstly, it was not possible to use streaming interfaces, that would have allowed to create a dataflow implementation on our FPGA. It was possible to use only Master-Slave interfaces that forced us to adapt our kernel implementations to this feature. The problem with the communication interfaces involved the difficulty of achieving the maximum bandwidth available. Hence we applied a methodology to compress the data that allowed to move a greater amount of data in a lower amount of time.

I have been able to test and benchmark the SDAccel tool-suite during my 6-month internship in Xilinx. During this period, I have been exposed to the bugs and limitation of this tool and I have been able to find new problematics related to it. Then, I helped the SDAccel team in California in reproducing the bugs so that they could be solved. In the meanwhile, I had to find turnarounds to the problems I was experiencing. Finally, I developed a test case, that will be described later in the document, for demonstrating the validity of the tool in real life applications.

CHAPTER 6

CASE STUDY

Here we will describe how we accelerated the two test cases introduced in Chapter 2. For each algorithm, a static code analysis is performed in order to estimate the performance of the system. Then, different optimizations are applied. The chapter provides information concerning the architectural choices that have been made as well as their justifications.

6.1 Hardware Implementation

The SDAccel framework has been used to design, optimize and accelerate the two test cases that we presented. Using the IDE, we have been able to import the code of the applications, analyze it, optimize it and emulate it performing a C simulation and a co-simulation. Then we used the tool to synthesize the optimized code and integrate it into the target FPGA. Note that this is a one-step operation.

Now we will present the implementation choices that we made in order to obtain the best architecture for the FPGA.

The first implementation choice, that concern both the applications, has been to code our algorithm using C/C++. This has been made in order to leverage the high abstraction of the code and the greater maturity of the tool in managing C/C++ code, rather than OpenCL. In fact, Vivado HLS, that is under the hood of SDAccel, is a state of the art software, present on the market since many years.

The choice of using C/C++, rather than OpenCL, has revealed also necessary. As SDAccel is still under development, the tool still has only preliminary support for OpenCL and not all its features are available yet.

Hence, we were not able to express the full potentiality of the tool by coding in OpenCL.

6.1.1 Smith-Waterman

The Smith-Waterman algorithm is the design I have been doing in my 6-months internship at the Xilinx Research Labs in Dublin. We have chosen to develop this algorithm due to its high parallelization possibilities. It is, in fact, a well know problem in the state of the art [14,15,21,24,26,27] . Many companies, as Rivyera [24] or Edico [61] developed cluster in order to efficiently perform genome sequencing based on this algorithm. These implementations involve the use of both FPGAs, GPUs and co-processors. The company Altera, main competitor of Xilinx, developed a version of the Smith-Waterman algorithm [28] written in OpenCL.

6.1.1.1 Preliminaries

The first step taken in the parallelization of this algorithm has been a static analysis of the code. This type of analysis provides us with information regarding what is the *complexity*, in terms of operations that this algorithm requires. The *static code analysis* is performed by counting how many *relevant* operations are performed by the kernel. As relevant operations, we counted three category of instructions: *Indexing*, *Arithmetical* and *Comparison*. Indexing instructions are all the operation concerning the calculation and the use of indexes, control operation are all the instruction that are involved into *if*, *for*, *while*, (etc...) constructs. Finally, arithmetical operation are the ones where there is a calculation of a result.

By just analyzing the code, we understood that there is only one part of the algorithm that is necessary to accelerate. In fact, it is not worth to parallelize the trace-back step of the computation, as it is strictly sequential.

In Table III, Table IV and Equation (Equation 6.1), it is possible to observe the static code analysis performed for our version of the Smith-Waterman algorithm. Table III depicts the type of operations with the theoretical formula and an example. Table IV express the traffic of memory from and to the device in terms of Bytes. Finally, Equation (Equation 6.1) shows the operational intensity by means of, how many operations are necessary per each byte of information that is processed.

TABLE III: SMITH-WATERMAN: STATIC CODE ANALYSIS - OPERATIONS

Work [Operations]	Theoretical	Example 256 x 65536
Indexing	$11N^2 + 11NM - 6N$	185M
Comparison	$6N^2 + 6NM - 5N$	101M
Arithmetic	$15N^2 + 15NM - 6N + 8M + 2$	253M
Total	$32N^2 + 32NM - 17N + 8M + 2$	539M

$$O.I. = \frac{32N^2 + 32NM - 17N + 8M + 2}{65N + 65M - 64} \tag{6.1}$$

It is clear, from the analysis, that the algorithm is *compute intensive* due to the amount of operations that it needs to perform. There are little reads, followed by massive writes. In fact,

TABLE IV: SMITH-WATERMAN: STATIC CODE ANALYSIS - MEMORY

Work [Operations]	Theoretical	Example 256 x 65536
Data In	$N + M$	65K
Data Out	$64(N + M - 1)$	4.2M
Total	$65M + 65M - 64$	4.3M

it has to store back the entire trace-back matrix. In the particular example provided, the OI (Operational Intensity) is 126 Operations per Byte but, tuning the values of N, the size of the query, and M the size of the database, it is easily discoverable that the OI ranges from 0.29, and moves to the *compute bound* area of the roofline model with $N = 1040$ and $M = 65536$.

The value of the operational intensity has then been used in the roofline model of the Alpha Data card (ADM-PCIE-7V3) to identify in which area was our algorithm. Thanks to the roofline model, we have been able to check what kind of performance to expect from this algorithm on the particular target board.

Figure 25 shows the position of our Smith-Waterman implementation on the roofline model of the Alpha Data board. It is clear that the algorithm is in the *memory bound* area, hence our performance are limited from the bandwidth capacity of the board.

In the state of the art of this algorithm, it is noticeable that the performance are calculated as *Giga Cell Update per Second*, or *GCUPS*. In Table III is observable as, in order to update one cell, the algorithm needs 32 operations. Therefore, the theoretical best performances achievable, with the example dataset, on the alpha data board are: $\frac{1024GOPS/s}{32ops} = 32GCUPS$.

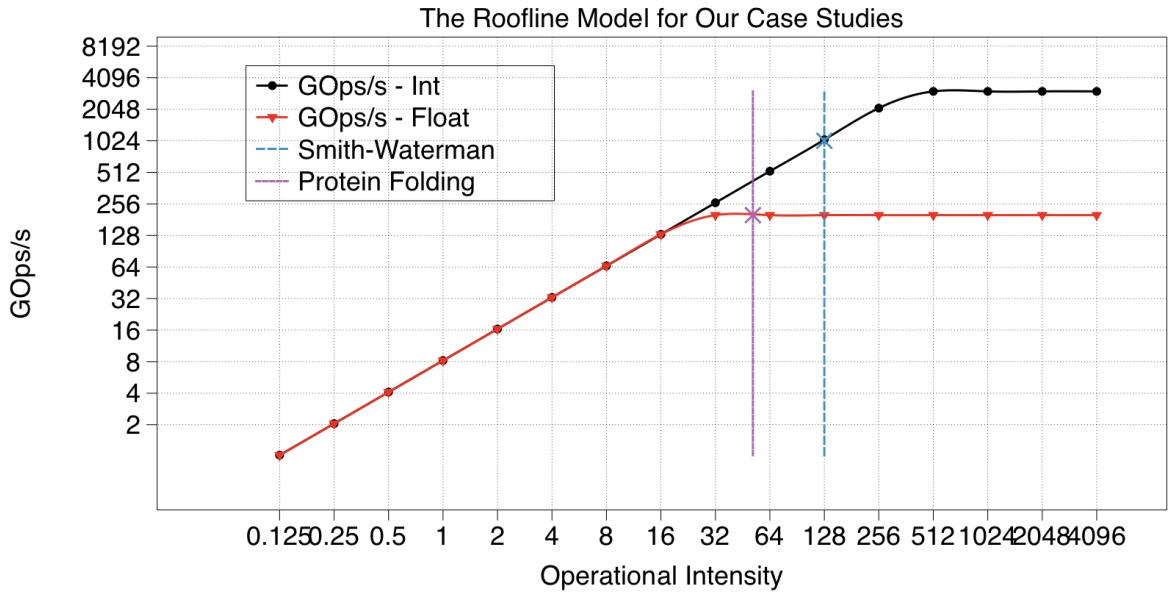


Figure 25: The Roofline Model of the Alpha Data Board - Case Studies

6.1.1.2 Implementation Features

It was clear from the description of the algorithm in Chapter 2, that the parallelism in this algorithm is on the anti-diagonal. In fact, each element on a anti-diagonal is dependency-free, given the values of the two previous anti-diagonals. Hence, we decided to exploit this thing creating an architecture performing parallel computation along the anti-diagonals using a *systolic array* that is column-based rather than diagonal-based. This is a key simplification that allowed us to achieve our goal: calculating all the elements that lies on a diagonal in one clock cycle. Coupled with this type of computation model we decided to buffer out the corners of the matrix in so that all the control logic necessary to understand how many, and which elements to compute is removed. It is clear that, in this way, the systolic array is computing

always the same amount of elements, N. However, although the kernel will have to compute a higher number of elements, the control logic has been simplified and then the computation is smoother.

Thanks to the information of the roofline chart (Figure 25) we found out that, on the Alpha Data board, we are memory bound. Hence we decided to use a *compressed layout*. The first compression that we performed is on the data. The two strings required for the computation are sequence of proteins, hence the elements that compose the sequence can have 4 possible values: A, C, G and T. We need only two bit of information in order to represent 4 values. Furthermore, we compressed in a 512 bit data structure 256 elements with this representation, highly diminishing the traffic of data from and to the host computer. Compressing the input data allows to increase the operational intensity. Raising this value, it is easier to obtain compute intensive kernels, and this is our objective as the roofline reflects that the kernel is in the memory bound area. The second compression is in the architecture itself. Whit this we mean that we are not translating the sequences in characters inside the kernel but we are directly using their 2-bit representation. Note that by using 512-bit structures to host the data, and using a 512-bit interface port, we can move 256 characters of each memory transaction.

Finally, we found out that, computing one anti-diagonal each clock cycle, we do need to store the entire query sequence, however we do not need to store the entire database. This because each time we calculate one anti-diagonal, we do need only N elements of it. Hence, we implemented a shift register with the same size of the compute window. The program shifts

in one database character every clock cycle so that the computation of a new anti-diagonal is performed.

As it is not necessary to store the entire Similarity matrix, the program keeps track in three buffers only of the values concerning the anti-diagonal that is being computed now and the two previous, for the dependency values. All the other elements of the matrix are then trashed. The device returns to the host program N values of the traceback matrix each clock cycle.

Once the computation is finished, the host program has received all the values of the traceback matrix, hence it can perform the final steps of the algorithm.

Now we will go through all the optimization steps that carried us to get the final results.

6.1.1.3 Systolic Array and Data Compression

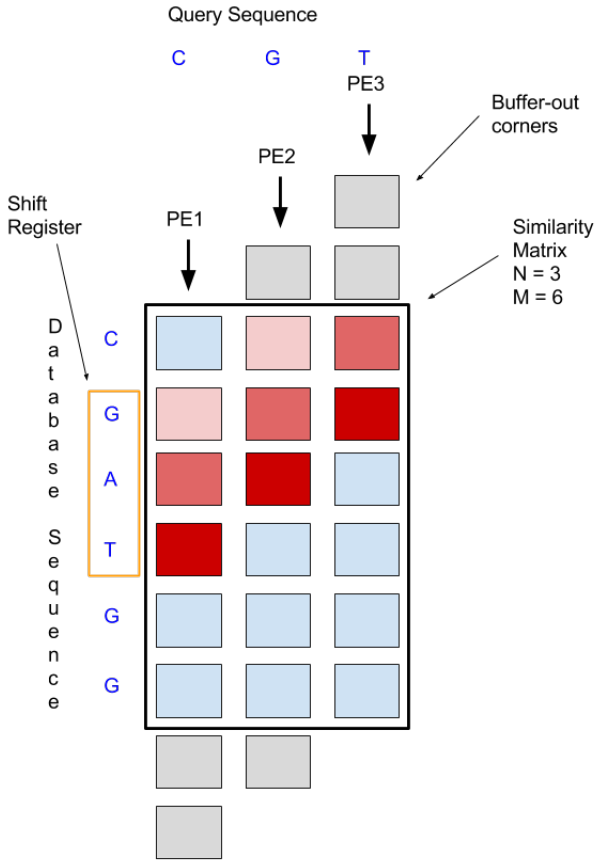


Figure 26: Systolic Array example with 3 Processing Elements

The first optimizations, that allowed us to break the wall of the Giga CUPS, are the implementation of the Systolic Array and the application of data compression on the output buffer.

The implementation of the systolic array of processing elements allowed us to perform a parallel computation along the anti-diagonals. Our version is column-based rather than diagonal based as, in this way, it is easier to handle the processing.

In Figure 26, is observable an example of similarity matrix calculation using a column-based systolic arrays composed of three PEs (Processing Elements). In the example, the size of the query is of 3 characters while the database is composed of 6. Hence the shift register (drawn in orange) will need to store only 3 elements each clock cycle. The three red anti-diagonals represent the three buffers that are being used in the calculation of the anti-diagonal, in darkest red. The others are the values calculated in the previous iterations and used as dependency values to calculate the new ones. Finally, the cells in gray are the one that are not directly used in the computation of the similarity matrix, but are necessary as we removed the control logic of the corners.

Data compression on the output buffer consist in expressing each element of the trace-back matrix using 2 bits. This representation allows a compression of 256 elements on a 512 bit particular structure available in SDAccel.

6.1.1.4 Shift Register and 512-bit Interfaces

After the first optimizations, we needed to increase the interface of the port communicating with the external of the kernel in order to fully utilize the 512-bit structure used, and to max the memory bandwidth utilization. We did that by increasing the port size to 512 bit so that each clock cycle, the kernel would be able to store 256 elements instead of only one. Furthermore, we noticed that by increasing the amount of data that the kernel was processing, the overall

performance of the system increased. For this reason, we decide to increase the amount of data processed, by increasing the size of the database string. Unfortunately, this raised a problem. Some of the buffers for handling the data inside the kernel are partitioned. Hence they are implemented as logic inside the hardware circuit. When a designer specifies an array as partitioned, the compiler will connect each logic element to all the possible paths that it could do, increasing drastically the complexity of the logic circuit. We solved this problem by implementing a shift register (Figure 26) for the database sequence. At the beginning of the computation, the kernel will copy only the first N elements of the database into a temporary register. Each clock cycle, a new element from the database will be shifted into the temporary buffer and a new computation step will be performed. By performing this operation, the size of the partitioned buffer is statically determined and fixed to the size of the query string, namely N. Note also that now the kernel could be able to handle database size of any dimensions.

6.1.1.5 Input and Layout Compression

Figure 27: Input Compression

By performing the previous optimizations, we were still missing something to get the maximum performances indicated by our roofline model. In particular, we noticed that the kernel was slowed down by the reading of the input variables. For this reason, we decided to apply data compression to the input strings too. Applying the compression, the input strings, raised the problem of having a decompression mechanism on the kernel so that the strings can be recognized by the program. Clearly, having a decompression mechanism resulted in a increase in the complexity of the hardware design, and the compiler beneath was not able to synthesize the circuit. Hence, we decide to not decompressing the input variables. Keeping a compressed layout, resulted in an increase of performance and a reduction of the complexity of the circuit as we were handling data considering a reduced bitwidth.

6.1.1.6 Multiple Memory Interfaces

The last optimization we made has been possible as we picked another board supported by the SDAccel framework. Hence we compiled the Smith-Waterman algorithm for the Kintex Ultrascale. On this board, two memory ports are implemented, and SDAccel, permit the user to define in which area of the global memory store buffers and on which port to send/receive the data.

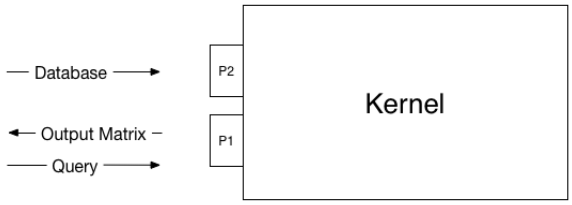


Figure 28: I/O Mapping on Kintex Ultrascale

We benchmarked two layouts of the Smith-Waterman on this board. The first version, is the same version we were testing on the Alpha Data, while the second one we exploited the two memory ports implemented on the Kintex by mapping one port to the first input and the result, and the second port to the second input.

The results of this optimizations will be explained in Chapter 7.

6.1.2 Protein Folding

This second case study, has been developed at Politecnico di Milano after the Smith-Waterman algorithm. This algorithm has been chosen to demonstrate how SDAccel can improve the productivity, in terms of development time, of the designer. In fact, this algorithm has been accelerated in only 3 weeks and, part of this work, resulted in a publication at the RAW conference 2016 [62]. The state of the art surrounding this algorithm is much smaller with regards to the previous; however, accelerations of algorithms for the prediction of the tertiary structure of proteins have been done. Jain et al. [39] proposed a hardware/software co-design, obtained by moving the most compute intense part of the algorithm into hardware. In [40], Zhu et al. pro-

posed a parallel architecture for the Backbone DBN program; an algorithm used for sampling the conformation of realistic proteins.

6.1.2.1 Preliminaries

This algorithm has been analyzed similarly to the Smith-Waterman algorithm. The first step taken in this analysis has been the identification of the most compute intense function. In order to do this, we used the profiling tool *Callgrind* paired with the visualization tool *KCachegrind*. Thanks to this tools, we have been able to identify the function *computePairEnergy* as the most compute intense. In fact, it occupies the 72.19% of the computation time.

Then, we had to translate and simplify the function. This step had been necessary as the code of this algorithm was written using dynamic memory allocation in C++. Clearly, dynamic memory allocation is not supported in HLS tools as Vivado HLS, hence we had to translate the code in a more static version of C/C++.

Once the code has been simplified, we performed a static code analysis in order to understand what kind of performances to expect from the code we would have run on the FPGA board. Here, the static code analysis has been more challenging as the complexity of this code depends on 4 different parameters: the two sizes of the proteins considered, indicated in Equation (Equation 6.2), Table V and Equation (Equation 6.3), *s1 (size1)* and *s2 (size2)* and the 2 values indicating the number of chemical groups of the side-chains of the proteins, called *ng1 (numgroup1)* and *ng2 (numgroup2)*.

The total number of operations, can be calculate by using the following formula:

$$s2(153s1 + 362) + 4s1 + ng1(3 + ng2(8 + 4s1 + 26s1s2)) + 139 \tag{6.2}$$

that, in our example result in 87K operations.

TABLE V: PROTEIN FOLDING: STATIC CODE ANALYSIS - MEMORY

Work	Theoretical	Example 1
Data In	$8(s1 + s2) + 4(ng1 + ng2) + 1470$	1.7K
Data Out	4	4
Total	$8(s1 + s2) + 4(ng1 + ng2) + 1474$	1.7K

$$\frac{s2(153s1 + 362) + 4s1 + ng1(3 + ng2(8 + 4s1 + 26s1s2)) + 139}{8(s1 + s2) + 4(ng1 + ng2) + 1474} \tag{6.3}$$

Equation (Equation 6.3) shows the final formula for the calculation of the code complexity for the protein folding. The bigger dataset we tested has, as input, the two sizes (s1 and s2) equal to 12, the groups number (ng1 and ng2) equal to 4 and the final value of the operational intensity equal to $51.58 OPS : B$.

As it is observable from the roofline in Figure 25, we are in the floating point domain and, with this particular operational intensity, our kernel is *compute bound*, hence we can optimize the code to the point of saturating the computational capabilities of the board. As explained

before, thanks to the roofline model we know what kind of performances we should expect from the kernel. Hence, with an operational intensity of 51.58 OPS:B we should have performances in the order of 200 GOps/s.

6.1.2.2 Implementation Features

From a first look at the code, it is clear that it is loop intense. The input variables pass among different loops in order to produce the single output. Given this introduction, we started optimizing the code applying *pragmas* to the code, aiming at reduce the latency of the loops. The first optimization that we performed has been to *pipeline* as many loops as possible. It has not been possible to pipeline all the loops present in the code as each time a pipelining on a loop is performed the amount of area required for the algorithm increases considerably. Note that a pipeline pragma in SDAccel unrolls the loop and the ones contained in the loop body.

Now we will explain the optimizations introduced.

6.1.2.3 Optimization I: Input Compression

The first optimization we applied to our kernel, is the compression of the input data into two big buffers. The code presents a high unbalance between the input data and the output one. Passing a considerable amount of input variables to a hardware kernel is not a good way of coding as the inputs could be stored in area of the memory that are not consequent, resulting in a decrease of performances. For this reason, we decided to optimize further the kernel by compressing all the inputs into two main buffers. The difference between this two buffers is that the first one holds only data that is *integer*, while the second handle *float-type* data. The compression has been applied by creating two buffers that are able to host all the input of a

particular type. By performing this optimization, we are able to ensure that all the data of the same type, resided in consequent areas of the memory. In this way, the tool is able to recognize the sequentiality of the data and it can improve the speed of the movement of data by inferring *memory bursts*. A memory burst increases the performance because it does not require to ask for the address in memory each memory transaction. It will ask for the address in memory only once, and then will be able to copy all the data that follows that address.

6.1.2.4 Optimization II: Control simplification and Loop pipelining

The second set of optimizations that we performed on the kernel are the simplification of the control constructs and the pipelining of the majority of the loops. A simplification of the control constructs mostly concerns the nested ifs, as in the following example:

```
1
2 if(cond1){
3   if(cond2){
4     ...
5   }
6 }
```

Such construct, introduces two levels of control that are not suggested in hardware as they introduce two levels of multiplexers. It can be simplified as follow:

```
1
2 if(cond1){
3   ...
```

```
4 }
5 if(cond1 && cond2){
6   ...
7 }
```

Now, the compiler is able to identify the computation blocks removing the latency introduced by the two levels of multiplexers.

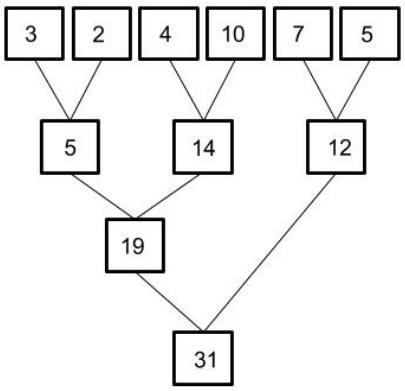
After this optimization, the majority of the loops have been pipelined. We pipelined the loops aiming at achieving an Initiation Interval of 1, so that the loop is able to produce a result each clock cycle. However, we have not been able to achieve II of 1 to each loop we pipelined as in the code there were many dependencies that did not allowed such optimization. Furthermore, it has not been possible to apply the pipeline pragma to all the loops present in the code, due to the physical limitation in area of the target board.

6.1.2.5 Optimization III: Reduction

One of the problems we incurred in when trying to achieve initiation interval (the number of clock cycles between the start time of two consecutive iterations of a loop) of 1 on loop pipelining is that, in this version of the protein folding algorithm, the output value is increased multiple times before it is returned to the host platform. Hence, there is a data dependency, in particular a *Read after Write (RAW)* on the output value. Our solution to this problem has been to apply a *reduction* to this part of algorithm [63]. The reduction works as follow.

Considering a multiple sum over the same variable as this:

```
1
```




```
1
2  int sum[N];
3  int sum_2[N/2];
4
5  for(int i = 0; i < N; i++){
6    sum[i] += function(parameter);
7  }
8
9  for(int i = 0; i < N/2; i++){
10   sum_2[i] = sum[2*i] + sum[2*i
       + 1];
11 }
12
13   ...
```

Figure 29: Example of reduction

```
2  for(int i = 0; i < N; i++){
3    sum += function(parameter);
4  }
```

The variable *sum* is increased each iteration of the loop, hence it is impossible to achieve II (Initiation Interval) of 1, once the loop pipelining pragma is applied. However, by creating an array capable of hosting N elements we would be able to achieve II = 1, given that *function* is able to produce results in time.

In order to perform such architectural optimization, we need the highest exponent x so that $2^x \geq N$, so $\lceil log_2 N \rceil$ array buffers.

The left side of Figure 29, shows an example of a reduction over six sums, while the right one shows an example of code after the reduction has been applied. The method works as

follow. Let's say our code presents 6 sums overs the same variable. This means that our first temporary array will have to be able to host six elements. Then, we change the code as in Figure 29 on the right side, so that each call of the function stores its results in a new slot in the temporary buffer. This removes the dependency of accumulating various results over the same variable, allowing the tool to efficiently pipeline the loop. At this point, we need to sum together the elements in the first temporary buffer, in order to have the final result. Hence, we create a new temporary buffer of size N/2 that sums together each two consecutive variables of the first buffer as in the left side of Figure 29. This process needs to be iterated until we sum only two elements, resulting in the final sum. Hence, it takes $log_2(N)$ steps to complete.

The reduction brought to the code great benefit. In fact, all of the loops that were slowed down from the dependency on a variable, have been able to get II of 1.

CHAPTER 7

EXPERIMENTAL EVALUATION

In this section we will discuss the results of our two case studies presented in Chapter 6. We will begin describing the experimental settings in Section 7.1, here we will introduce the testing environment we created in order to measure the performance of our applications and the board used to run the algorithms. Then we will show our results and we will compare them with implementations of the same algorithms available in the state of the art.

7.1 Experimental Settings

The two applications proposed here have been optimized and compiled using the SDAccel framework. The host computer is a x64 machine running Red Hat Linux Enterprise 6.6. The FPGA board is connected to the host via PCIe and has been configured to support the SDAccel framework [64].

The execution times for the implemented kernels have been measured by exploiting the *events* of the OpenCL language. SDAccel provides with a detailed report after each execution of the kernel that specifies the execution time of the algorithm on the FPGA board.

The software comparison of the Protein Folding Algorithm has been made with an Intel i7 running at 2.7GHz. Here the execution time has been measured using built-in C++ functions.

7.1.1 Smith-Waterman

The Smith-Waterman takes as input a query string, a database and a scoring system. Our benchmarks have been done using random query and database strings. The scoring system is as follow: each gap is scored as -1, the score given for a match is equal to $+2$, while the one for a mismatch is -1. In the version of the code benchmarked here, the calculation of the max-score has been left out. However, this calculation does not impact performance as it does not complicate the logic of the circuit.

7.1.2 Protein Folding

The *computePairEnergy* function of the protein folding algorithm, takes as input all the information concerning two proteins and return the final sum of energy.

In order to test this algorithm, we took two test cases. The first test case, is the intermediate result of an execution of the protein folding algorithm with an example input. The second one, is a random generated example. The size of the input is variable, as it depends on the dimension of the proteins considered. The second input we tested our code with, involves bigger proteins than the first one, hence the calculation performed are more.

The version of the algorithm we have accelerated has been taken from [33].

7.1.3 The AlphaData ADM-PCIE-7V3 Card

The Alpha Data *ADM-PCIE-7V3* is a board produced by the company Alpha Data that is based on the Xilinx *Virtex-7* (XC7VX690T-2) FPGA device. It is a reconfigurable board targeted for high performance computing. It's applications range from HPC and Data Centers, to Network Accelerators and many others. [65]

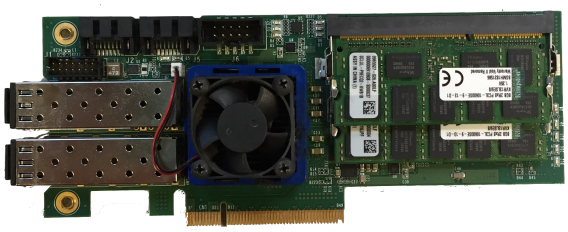


Figure 30: The Alpha Data ADM-PCIE-7V3 Board

The ADM-PCIE-7V3 is one of the board available within the SDAccel framework and, in this section, we are going to provide with an overview of the structure of this board.

The block diagram, showing the architecture of the board is visible in Figure 31. In the picture, we can observe that the central part of the diagram is occupied by the FPGA. It is a *Xilinx Virtex-7* FPGA, in particular, the model is XC7VX690T. The FPGA is interfaced with a *x8 PCIe Gen 3* with a variable number of lanes that can be 1, 2, 4 or 8. The maximum speed of this link is 8 GB/s when used with 8 lanes.

Two DDR3 SDRAM SODIMM connectors are available on the board and can accommodate up to two slots of memory with maximum signaling rate of 1333 MT/s and a maximum capacity of 8 GiB per slot.

On the left side of the block diagram there are 2 SFP+ cages that allow communications up to 10G Ethernet or also other protocols that are compliant with the Xilinx MGT Transceivers.

Internal SATA management can be handled by the two receptacles supporting third generation specification working at 6 Gbps.

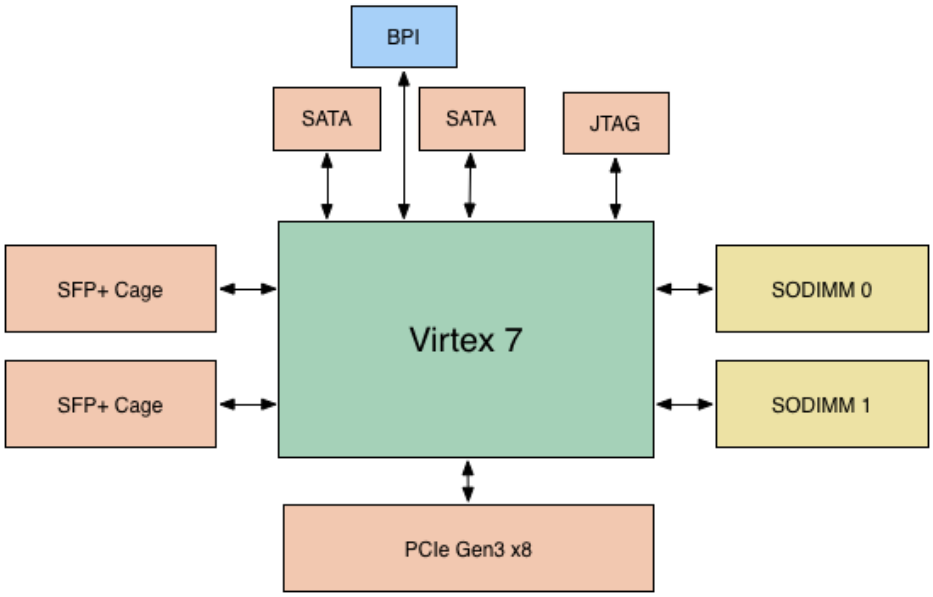


Figure 31: The Alpha Data ADM-PCIE-7V3 Board Block Diagram

The board is equipped with a system monitor too. The task of the monitor is to check the voltage, temperature, power and current values and act if there is some fault in the system. The user is allowed to read this values from a PC by using the PMBUS.

Finally, in order to allow a configuration of the device on the board, there is a JTAG or it is possible to use the BPI flash memory device to boot the configuration at power on.

7.1.4 The Kintex Ultrascale

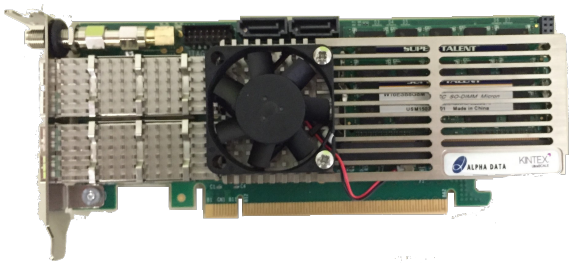


Figure 32: The Alpha Data ADM-PCIE-KU3 Board

The Alpha Data ADM-PCIE-KU3 is another board produced by the company Alpha Data, based on the FPGA Xilinx Kintex Ultrascale (XCKU060). As the previous board, it is a re-configurable device targeted for high performance computing, with an applicability that ranges from HPC and Data Centers, to Network accelerators and others.

This board is one of the possible targets in the SDAccel framework and we will go through its general architecture in the next section.

Figure 33 shows an overview of the architecture of the board. In the center, there is the Kintex Ultrascale FPGA that is interfaced with 2 *x8 PCIe Gen 3* with a number of lanes that can be 1,2,4,8 or 16. The SODIMM memory connectors, as well as the SATA ports, are the same

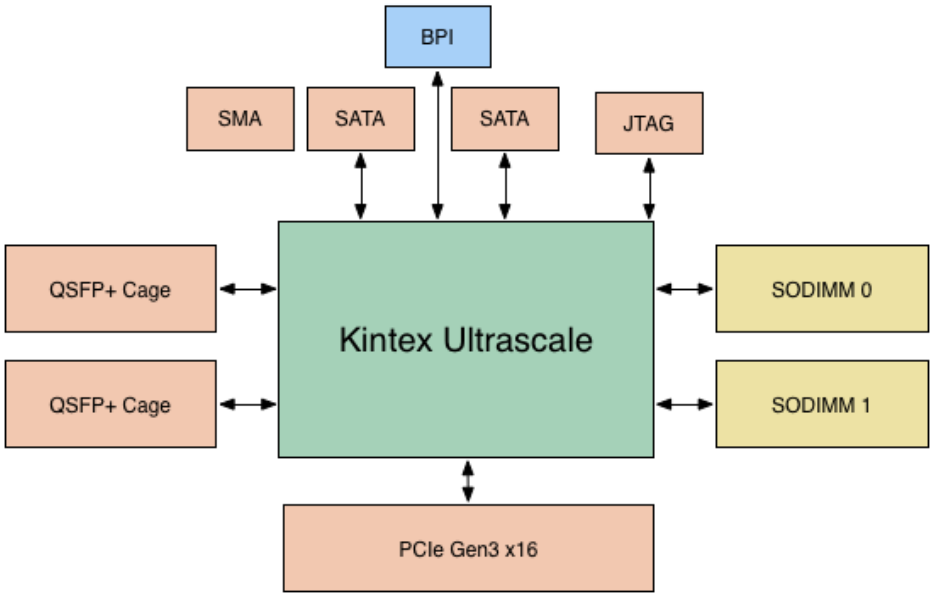


Figure 33: The Alpha Data ADM-PCIE-KU3 Board Block Diagram

as in the previous board. There are 2 *QSFP+* cages for Ethernet connection, or connection by Xilinx-supported protocols.

There is a system monitor onto this board too. It allows to monitor voltage, power, current and temperature. It also checks and acts, in case there is a fault in the system. Finally, as in the previous board, the device is configurable using the flash memory or the JTAG port.

7.2 Results and Comparisons

Here we will present the results, that we achieved with our implementations of the two algorithms. The results will then be compared with state of the art implementations' results by mean of comparing the execution times of the different implementations.

7.2.1 Smith-Waterman

The results of this algorithm are expressed as GCUPS - Giga Cell Update per Second - as explained in the previous chapters. The performances have been achieved by increasing the number of optimizations performed. Furthermore, we have been able to improve performances also by changing the way some optimization has been performed. Thanks to the roofline model, we had information about how much we would have been able to optimize the kernel on the specific platform.

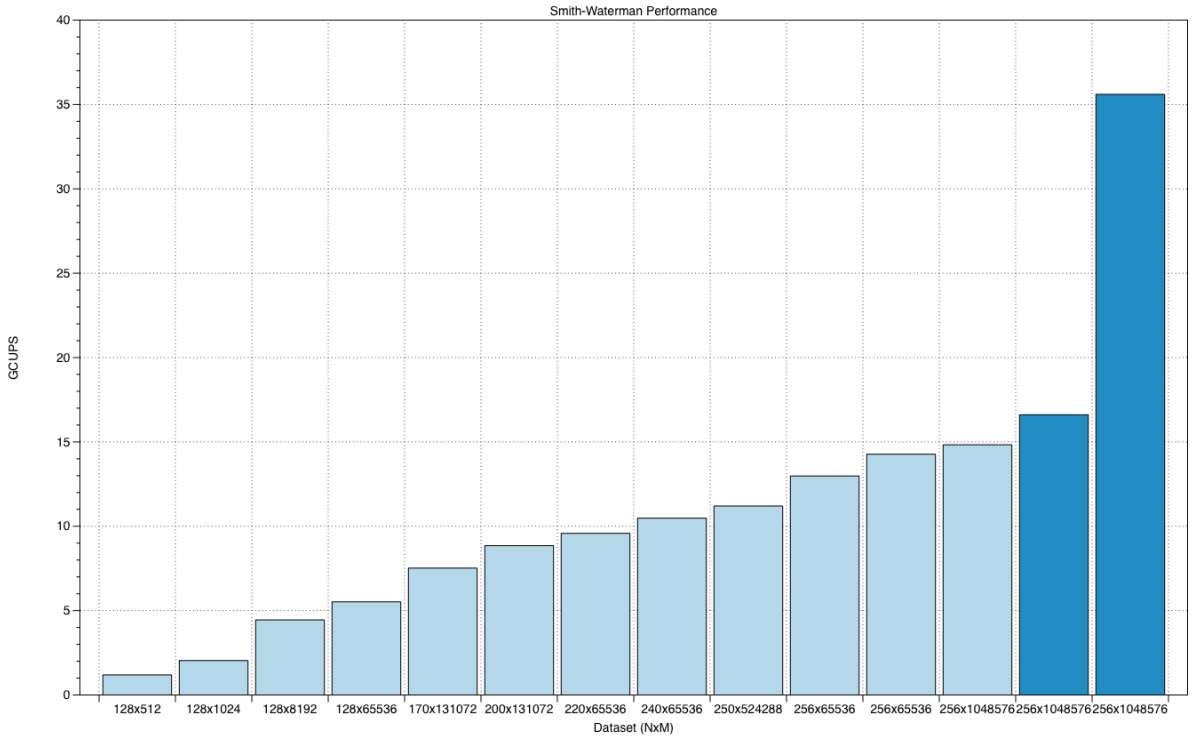


Figure 34: Smith-Waterman Performance

Figure 34, shows the performance of our Smith-Waterman kernel. On the x-axis there are the datasets, by means of query size (N) and database size (M), that obtained the result, in GCUPS, expressed on the y-axis. It is noticeable how, the performance of the algorithm is highly dependent on the query size (N). In fact, the best performances have been achieved with a query size of 256. This is because, as the parallelism is on the anti-diagonal and our implementation works by computing a anti-diagonal each clock cycle, augmenting the size of the diagonal (hence the dimension of the anti-diagonal), we are updating more cell in parallel each clock cycle. However, the maximum number of elements that we have been able to execute in parallel is 256. After this number, the logic of the circuit we have implemented was too complex to synthesize. From the other hand, our implementation of the algorithm is not dependent on the size of the database. In fact, thanks to the shift register the size of the database does not complicate the logic of the implemented circuit.

Thanks to the first optimizations, concerning the implementation of a *systolic array* of processing elements and *data compression*, we have been able to reach a bit more of a Giga CUPS. Then we implemented the shift register, in order to increase the possible size of the database used. Thanks to this, we have been able to process more data, exploiting the capabilities of the FPGA in managing a high amount of data. This, with the increase of port size to 512 bit allowed us to get up to 11 GCUPS by using 250 elements as query size.

The third set of optimizations performed, has been the *input and layout compression*. However, it allowed us to reach roughly 15 GCUPS, less than half of the values predicted by the roofline model. This is because on the Alpha Data card, has only one 8GB/ s memory port

hence, all the communication from and to the kernel are multiplexed on that port. Then, it means that when write and read happen simultaneously the channel is shared and gives 4GB/s of bandwidth for each direction. Hence, with our 15 GCUPS implementation we are maxing out the bandwidth, and this is the maximum achievable performance on the Alpha Data card with SDAccel 2015.1.

The last choice we made, has been to move to the Kintex Ultrascale. The first version of the kernel, as explained in the previous chapter, allowed us to get the results observable in Figure 34 on the first darker blue bar. With this new board, the version that was achieving 15 GCUPS on the Alpha Data is now able to get more. The reason why the Kintex Ultrascale is able to get more performances is that it can exploit a higher bandwidth capacity of roughly 10 GB/s. The second bar, shows the performance after the port mapping optimization. The reader may observe that we passed the 35 GCUPS, in particular we obtained 35.6 GCUPS.

7.2.2 Comparison with State of the Art

In this section we will present how, state of the art algorithms compare to our implementation. The table in Table VI shows the performances of the algorithm that really compare to us, hence the one that are confronting one query sequence against one database sequence.

The values for calculating the ratio $performance/power\_consumption$ has been taken from the specification on the vendor websites. For what concerns the ratio of the Altera's implementation on the Nallatech board, that one has been taken from this paper [28].

From Table VI, it is clear that the best ratio performance over power consumption is achieved by the Altera Stratix V on Nallatech PICEe-385. However, our implementation, not only is

TABLE VI: SMITH-WATERMAN: PERFORMANCE IN STATE OF THE ART

Platform	Performance[GCUPS]	Ratio [GCUPS/W]
2 x NVidia GeForce 8800 [16]	3.6	0.017 [66]
single-GPU NVIDIA GeForce GTX 280Âă [18]	9.66	0.041 [67]
dual core Nvidia 9800 GX2 [15]	14.5	0.074 [68]
Nvidia GeForce GTX 295 [18]	16.087	0.056 [69]
Altera Stratix V on Nallatech PICEe-385 [28]	24.7	0.988 [28]
Nvidia GeForce GTX 295 [20]	30	0.104 [69]
ADM-PCIE-7V3	**35.6**	**1.424**
Tesla K20 [14]	45	0.2 [70]

better from a performance point of view, but also if we consider the ratio. In fact, the power consumption of this board is 25W [71], and the ratio performance over power consumption is 35.6/25 = 1.424. Actually, this values is not only better than the one achieved by Altera, it also greater than all the other considered.

7.2.3 Protein Folding

As explained above, the protein folding algorithm, and more in particular the compute pair energy function, has been tested by using 2 different datasets. We implemented three levels of optimization on this algorithm and we tested each of these optimizations with the small and the big dataset.

Figure 35 and Figure 36 show the result of the compute pair energy function on the two datasets. On the y-axis is represented the execution time in millisecond, while each bar of the chart is the result of an optimization. The first bar represents the naive version of the code, that is the first hardware implementation with no optimization applied. Then the three optimized

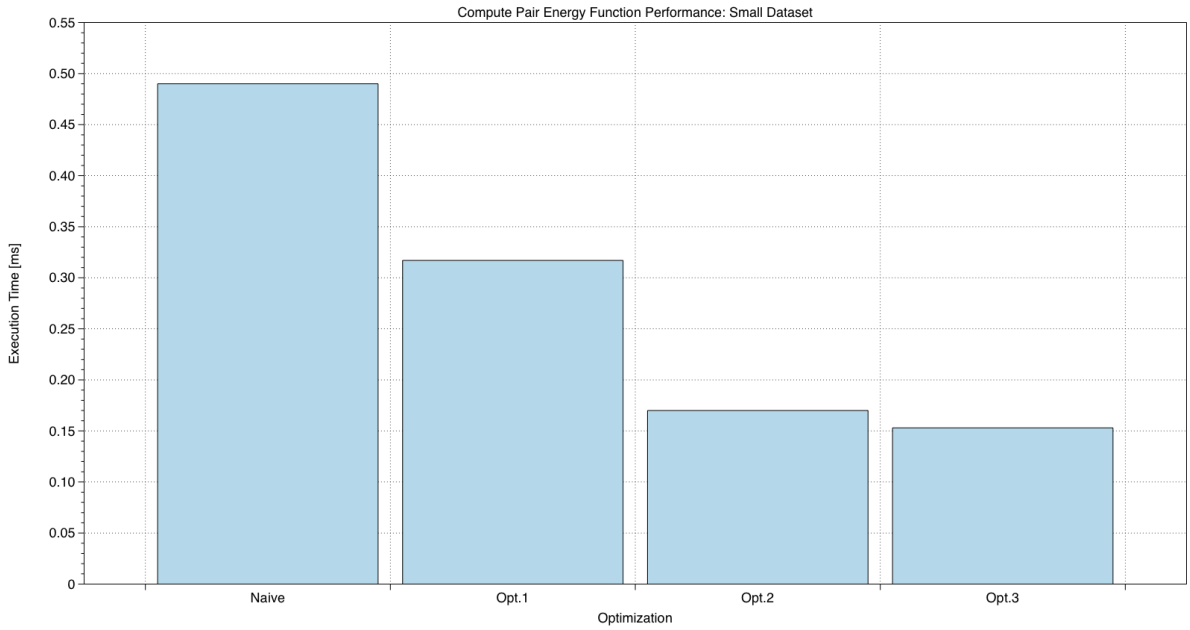


Figure 35: Compute Pair Energy Performance: Small Dataset

implementation that have been discussed in Chapter 6, namely the input compression, the control simplification and loop pipelining and, finally the reduction.

Table VII and Table VIII show the comparison between the performances of the CPU and the execution times on the FPGA board. The fourth column of the tables represent the speed-up achieved by the FPGA board. It is noticeable how, the best implementation has a speed-up of 1.61x with relation to the pure software implementation. The Hardware present a ratio of performance over power consumption that is greater. The values regarding the power consumption of the CPU has been taken from here[72]. We calculated the performances as

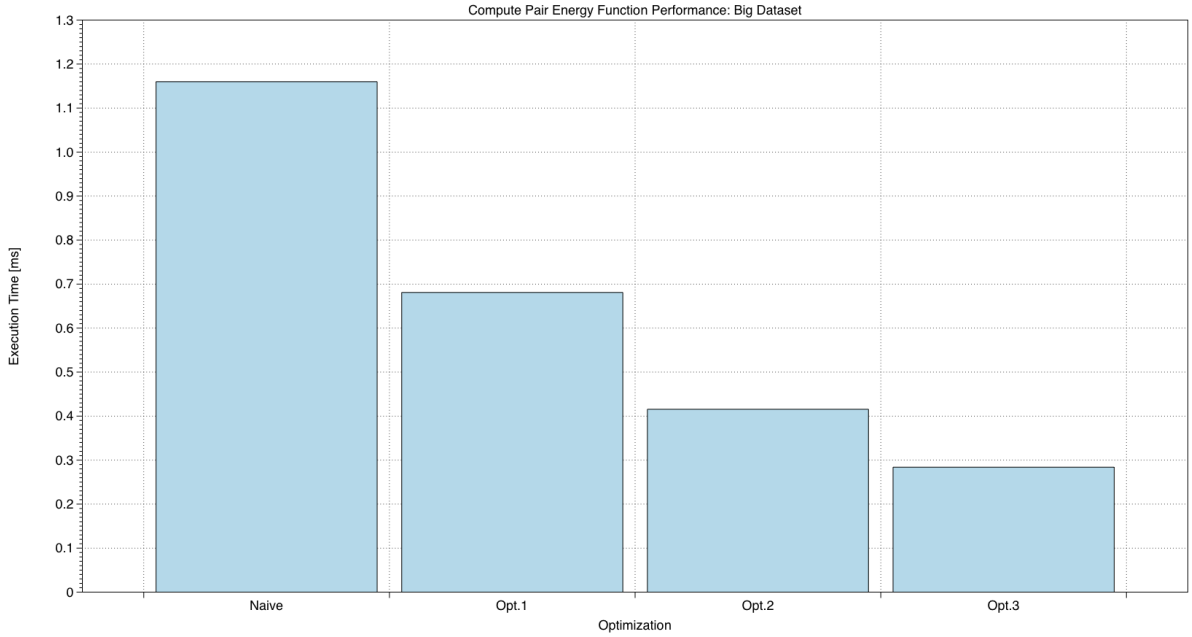


Figure 36: Compute Pair Energy Performance: big Dataset

$\frac{1}{execution_{time}}$, hence it is not possible to compare the ratio of the big dataset with the ratio of the smaller one as, in this formula, there is no contribution of the input dataset.

If, from the one side, the speed-up in terms of execution time is not very high, the one in terms of design time has been considerable. The time spent in profiling the application, porting the kernel code into SDAccel, optimizing it and building the application to run on the board has been only 3 weeks. This is a very good result as usually the hardware design process takes months to be performed.

TABLE VII: CPU VS FPGA: SMALL DATASET

$Version$	$Time_{FPGA}$ (ms)	$Time_{CPU}$ (ms)	$Speedup$	$Ratio_{FPGA}$	$Ratio_{CPU}$
Naive	0.490	0.2125	0.43	0.082	0.134
Opt1	0.317	0.2125	0.67	0.126	0.134
Opt2	0.170	0.2125	1.25	0.235	0.134
Opt3	0.153	0.2125	1.39	0.261	0.134

TABLE VIII: CPU VS FPGA: BIG DATASET

$Version$	$Time_{FPGA}$ (ms)	$Time_{CPU}$ (ms)	$Speedup$	$Ratio_{FPGA}$	$Ratio_{CPU}$
Naive	1.16	0.4425	0.38	0.034	0.065
Opt1	0.681	0.4425	0.65	0.059	0.065
Opt2	0.415	0.4425	1.07	0.096	0.065
Opt3	0.275	0.4425	1.61	0.14	0.065

7.2.4 Comparison with State of the Art

As said before, the state of the art regarding this version of the protein folding algorithm, is not so popular and the result are expressed only by means of comparison to CPU version of the algorithm.

In [39], the results the authors have achieved is a speed-up of 5x over a pure software implementation. The second parallel implementation, the one by Zhi et al [40], they compare their implementation with an Intel E7400, a dual-core CPU. The results they have been able to achieve are a speed-up of over 20x with relation to this platform.

Although our results are a speed-up of 1.61x relating to an i7 processor, it is hard to compare these results given the wide amount of algorithm for computing the protein folding that are available in the state of the art and the differences in measuring the performance.

CHAPTER 8

CONCLUSIONS AND FUTURE WORKS

We presented two examples of hardware accelerations using the SDAccel framework.

Using SDAccel to accelerate this two applications did bring many benefits to us. The environment offered by the framework revealed to be user-friendly and allow writing and optimizing of code in a easy-to-use environment, equipped with debugging and profiling tools to test and correct the application.

The first algorithm, is the result of an internship that I have done at the Research Labs of Xilinx Dublin. The result of the acceleration showed a performance of more than 35 GCUPS by using the Kintex Ultrascale board. In this test case, SDAccel revealed to be a good tool to design HPC applications, in fact as far as we know, our version of the algorithm presents the highest ratio performance over power consumption in the state of the art, compared to algorithms that pair one query string to one database.

The second implementation is taken from the biomedical field. It is a protein folding algorithm and has been developed at Politecnico di Milano. This implementation has been optimized and we obtained a speed-up of 1.61x with relation to a Intel i7 processor running at 2.6 Ghz. Although the performances are not so high, the development time of this algorithm has been very low. Thanks to SDAccel, in fact, it took us only 3 weeks to have the version we are presenting in this document, usually a design flow takes much longer. Furthermore, part of this work is part of a publication at the RAW conference[62].

8.1 Future works

For what concern the Smith-Waterman algorithm, the future works involve the integration of the max calculation in order to complete the algorithm. Note that this part of the logic does not increase the complexity of the logic circuit implemented so far. Furthermore, it is possible to increase the performance further by aligning more than a query to the same database string. The protein Folding algorithm has been implemented to demonstrate the speed-up that SDAccel offers, in terms of developing time, to the programmer. The design flow introduced by the tool, in fact, allows a fast developing environment. For this algorithm, we could work by increasing the level of optimization in the kernel trying to exploit the full parallelization capabilities of the FPGA board.

CITED LITERATURE

1. Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D.: Dark silicon and the end of multicore scaling. In Computer Architecture (ISCA), 2011 38th Annual International Symposium on, pages 365–376. IEEE, 2011.
2. Gnemmi, G., Crippa, M., Durelli, G., Cattaneo, R., Pallotta, G., and Santambrogio, M. D.: On how to efficiently accelerate brain network analysis on fpga-based computing system. In 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pages 1–6. IEEE, 2015.
3. Neuendorffer, S., Li, T., and Wang, D.: Accelerating opencv applications with zynq-7000 all programmable soc using vivado hls video libraries. Xilinx Inc., August, 2013.
4. Feist, T.: Vivado design suite. White Paper, 2012.
5. Cattaneo, R., Pilato, C., Durelli, G. C., Santambrogio, M. D., and Sciuto, D.: Smash: A heuristic methodology for designing partially reconfigurable mpsocs. In Rapid System Prototyping (RSP), 2013 International Symposium on, pages 102–108. IEEE, 2013.
6. Edwards, S. et al.: The challenges of synthesizing hardware from c-like languages. Design & Test of Computers, IEEE, 23(5):375–386, 2006.
7. Bobda, C.: Introduction to Reconfigurable Computing. Architecture, algorithm and applications. Springer, 2007.
8. Hsiung, P.-A., Santambrogio, M. D., and Huang, C.-H.: Reconfigurable System Design and Verification. CRC Press, 2009.
9. Williams, S., Waterman, A., and Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65–76, 2009.
10. Smith, T. and Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology, 147(1):195 – 197, 1981.
11. Smith-waterman algorithm. `http://cs.stanford.edu/people/eroberts/courses/soco/projects/computers-and-the-hgp/smith_waterman.html`. Accessed: 2016-03-09.
12. Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J.: Basic local alignment search tool. Journal of molecular biology, 215(3):403–410, 1990.
13. Pearson, W. R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. Genomics, 11(3):635–650, 1991.
14. Huang, L., Wu, C., Lai, L., and Li, Y.: Improving the mapping of smith-waterman sequence database searches onto cuda-enabled gpus. BioMed research international, 2015, 2015.

15. Ligowski, Ł. and Rudnicki, W.: An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases. In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pages 1–8. IEEE, 2009.
16. Manavski, S. A. and Valle, G.: Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. BMC bioinformatics, 9(2):1, 2008.
17. Munekawa, Y., Ino, F., and Hagihara, K.: Design and implementation of the smith-waterman algorithm on the cuda-compatible gpu. In BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on, pages 1–6. IEEE, 2008.
18. Liu, Y., Maskell, D. L., and Schmidt, B.: Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. BMC research notes, 2(1):73, 2009.
19. Hasan, L., Kentie, M., and Al-Ars, Z.: Dopa: Gpu-based protein alignment using database and memory access optimizations. BMC research notes, 4(1):261, 2011.
20. Blazewicz, J., Frohmberg, W., Kierzynka, M., Pesch, E., and Wojciechowski, P.: Protein alignment algorithms with an efficient backtracking routine on multiple gpus. BMC bioinformatics, 12(1):1, 2011.
21. Torbjørn, R.: Faster smith-waterman database searches with inter-sequence simd parallelisation. BMC Bioinformatics, 12(1):221, 2011.
22. Altera: Altera SDK for OpenCL - Programming Guide. Altera.
23. Hybrid computing for high-throughput bioinformatics. `http://www.pccluster.org/ja/event/wsinkobe/08_Nishi.pdf`. Accessed: 2016-03-11.
24. SciEngines: Accelerated smith-waterman on rivyera hardware. 2012.
25. Wozniak, A.: Using video-oriented instructions to speed up sequence comparison. Computer applications in the biosciences: CABIOS, 13(2):145–150, 1997.
26. Rognes, T. and Seeberg, E.: Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics, 16(8):699–706, 2000.
27. Farrar, M.: Striped smith–waterman speeds database searches six times over other simd implementations. Bioinformatics, 23(2):156–161, 2007.
28. Settle, S. O.: High-performance dynamic programming on fpgas with opencl. In Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC), pages 1–6, 2013.
29. Khoury, G. A., Smadbeck, J., Kieslich, C. A., and Floudas, C. A.: Protein folding and de novo protein design for biotechnological applications. Trends in biotechnology, 32(2):99–109, 2014.
30. Otto, R., Santagostino, A., and Schrader, U.: Rapid growth in biopharma: Challenges and opportunities. From Sci. to Oper. Quest. Choices Strateg. Success Biopharma, pages 1–21, 2014.

31. Lotan, I., Schwarzer, F., Halperin, D., and Latombe, J.-C.: Efficient maintenance and self-collision testing for kinematic chains. In Proceedings of the eighteenth annual symposium on Computational geometry, pages 43–52. ACM, 2002.

32. Lotan, I., Schwarzer, F., and Latombe, J.-C.: Efficient energy computation for monte carlo simulation of proteins. In Algorithms in Bioinformatics, pages 354–373. Springer, 2003.

33. Protein folding mcs - stanford. http://robotics.stanford.edu/~itayl/mcs/. Accessed: 2016-04-09.

34. Lazaridis, T. and Karplus, M.: Effective energy function for proteins in solution. Proteins: Structure, Function, and Bioinformatics, 35(2):133–152, 1999.

35. Eisenberg, D. and McLachlan, A. D.: Solvation energy in protein folding and binding. Nature, 319(6050):199–203, 1985.

36. Kim, D. E., Chivian, D., and Baker, D.: Protein structure prediction and analysis using the robetta server. Nucleic acids research, 32(suppl 2):W526–W531, 2004.

37. Cheng, J., Randall, A. Z., Sweredoski, M. J., and Baldi, P.: Scratch: a protein structure and structural feature prediction server. Nucleic acids research, 33(suppl 2):W72–W76, 2005.

38. Hung, L.-H., Ngan, S.-C., Liu, T., and Samudrala, R.: Protinfo: new algorithms for enhanced protein structure predictions. Nucleic Acids Research, 33(suppl 2):W77–W80, 2005.

39. Jain, A., Gambhir, P., Jindal, P., Balakrishnan, M., and Paul, K.: Fpga accelerator for protein structure prediction algorithm. In Programmable Logic, 2009. SPL. 5th Southern Conference on, pages 123–128. IEEE, 2009.

40. Zhu, Q., Xia, F., and Jin, G.: Fpga-based accelerator for sampling realistic protein conformations. International Journal of Advancements in Computing Technology, 5(8), 2013.

41. Guo, Z., Najjar, W., and Buyukkurt, B.: Efficient hardware code generation for fpgas. ACM Transactions on Architecture and Code Optimization (TACO), 5(1):6, 2008.

42. Buyukkurt, B., Cortes, J., Villarreal, J., and Najjar, W. A.: Impact of high-level transformations within the roccc framework. ACM Transactions on Architecture and Code Optimization (TACO), 7(4):17, 2010.

43. Villarreal, J., Park, A., Najjar, W., and Halstead, R.: Designing modular hardware accelerators in c with roccc 2.0. In Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pages 127–134. IEEE, 2010.

44. Fingeroff, M. and Gopalsamy, T.: Designing high performance dsp hardware using catapult c synthesis and the altera accelerated libraries. White Paper, 2007.

45. Catapult c synthesis. http://www.iuma.ulpgc.es/~nunez/clases-micros-para-com/clases-mpc-slides-links/Embedded-ARM-ST-o-Platforms-MPSoC/CatapultDataSheetWeb.pdf. Accessed: 2016-03-21.

46. Bollaert, T.: Catapult synthesis: a practical introduction to interactive c synthesis. In High-Level Synthesis, pages 29–52. Springer, 2008.

47. Pellerin, D. and Thibault, S.: Practical fpga programming in c. Prentice Hall Press, 2005.

48. Panda project. http://panda.dei.polimi.it/. Accessed: 2016-03-23.

49. Bambu framework. http://panda.dei.polimi.it/?page_id=31. Accessed: 2016-03-23.

50. Pilato, C. and Ferrandi, F.: Bambu: A modular framework for the high level synthesis of memory-intensive applications. In Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pages 1–4. IEEE, 2013.

51. De Dinechin, F. and Pasca, B.: Designing custom arithmetic data paths with flopoco. IEEE Design & Test of Computers, (4):18–27, 2011.

52. Altera's opencl sdk: High-level synthesis done a different way. http://www.bdti.com/InsideDSP/2013/02/13/Altera. Accessed: 2016-03-31.

53. Fpga cluster computing. https://www.andrew.cmu.edu/user/xiaoboz/15418/finalreport.html. Accessed: 2016-03-30.

54. The green 500. http://www.green500.org/. Accessed: 2016-03-30.

55. Sundararajan, P.: High performance computing using fpgas. Xilinx White Paper: FPGAs, pages 1–15, 2010.

56. Nvidia cuda. http://www.nvidia.com/page/home.html. Accessed: 2016-03-30.

57. Stone, J. E., Gohara, D., and Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(1-3):66–73, 2010.

58. Xilinx: The xilinx sdaccel development environment - bringing the best performance/watt to the data center. 2014.

59. Wirbel, L.: Xilinx sdaccel - a unified development environment for tomorrow's data center. November 2014.

60. Xilinx: SDAccel Development Environment - User Guide. Xilinx.

61. Edico dragen. http://www.edicogenome.com/dragen/dragen-bio-it-platform/. Accessed: 2016-03-28.

62. Guidi, G., Reggiani, E., Di Tucci, L., Durelli, G., Blott, M., and Santambrogio, M. D.: On how to improve fpga-based systems design productivity via sdaccel. In to be published, page to be published. to be published, 2016.

63. OpenclâĎć optimization case study: Simple reductions. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/. Accessed: 2016-04-12.

64. Xilinx: SDAccel Development Environment - Tutorial. Xilinx.

65. ALPHA DATA: ADM-PCIE-7V3 Datasheet, 11 2013. Rev. 1.0.

66. Nvidia geforce 8800 gt specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-8800-gt/specifications. Accessed: 2016-04-04.

67. Nvidia geforce gtx 280 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications. Accessed: 2016-04-04.

68. Nvidia geforce 9800 gx2 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-9800gx2/specifications. Accessed: 2016-04-04.

69. Nvidia geforce gtx 295 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295/specifications. Accessed: 2016-04-04.

70. Tesla k20 gpu accelerator - board specification. https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. Accessed: 2016-04-04.

71. ALPHA DATA: ADM-PCIE-KU3 User Manual, 1 2016. Rev. 1.8.

72. Intel core i7-2620m. http://www.notebookcheck.net/Intel-Core-i7-2620M-Notebook-Processor.40108.0.html. Accessed: 2016-04-04.

VITA

Lorenzo Di Tucci

Education

B.S., Engineering of Computing Systems
Politecnico di Milano
July 2013

M.S., Engineering of Computing Systems
Politecnico di Milano
2013-2015

M.S., Computer Science
University of Illinois at Chicago, Chicago, IL
2013-2015