POLITECNICO
MILANO 1863

Politecnico di Milano
Facoltà di Ingegneria Industriale e dell'Informazione
LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Exploiting FPGA from Data Science Programming Languages

Master of Science thesis of:
Luca Stornaiuolo
Matricola 864278

Advisor:
Ing. Marco D. Santambrogio

Co-advisors:
Dott. Ing. Emanuele Del Sozzo

Academic year 2016/2017

Acknowledgments

Cosí andammo infino a la lumera,
parlando cose che 'l tacere é bello,
sí com'era 'l parlar colá dov'era.
[...]
Io non posso ritrar di tutti a pieno,
peró che sí mi caccia il lungo tema,
che molte volte al fatto il dir vien meno.

Dante Alighieri, Divina Commedia, Inferno IV

Molte sono le persone che voglio ringraziare per aver permesso la realizzazione di questo lavoro. Un enorme ringraziamento va al mio relatore, Marco Domenico Santambrogio, che ha creduto in me e mi ha permesso di affrontare ogni giorno il mio "brick wall", accompagnandomi in un percorso di crescita sia professionale (con la possibilá di mettermi alla prova come studente di ingegneria e ricercatore), ma soprattutto personale, inserendomi in un contesto ricolmo di opportunitá, nel quale ho conosciuto persone meravigliose con cui imparare e condividere esperienze. Tra queste persone, Emanuele Del Sozzo, con cui, nel momento in cui scrivo la tesi, sto trascorrendo il periodo estivo oltre oceano nei pressi di Boston, mi ha aiutato nel difficile compito di scrittura e correzione di questo testo, con la giusta severitá e il giusto incoraggiamento, e mi é stato vicino in momenti non facili, aiutandomi a risollevarmi (letteralmente).

Durante la realizzazione di questo lavoro ho avuto il privilegio e la fortuna di collaborare con Alberto Parravicini, Gabriele Pallotta e Gianluca Carlo Durelli, per la parte di sistemi Desktop e con Riccardo Pressiani, Filippo Carloni, Giuseppe Natale e Sara Notargiacomo, per la parte di sistemi Embedded. Tutti hanno dato un enorme e prezioso contributo e il mio augurio é quello di poter lavorare ancora con loro in futuro. Inoltre voglio ringraziare i veterani (oltre i giá citati Emanuele, Gabriele, Gianluca e Giuseppe) Marco Rabozzi e Alberto Scolari sempre

pronti a condividere la propria saggezza e tentare di mettere ordine al caosss. Un enorme abbraccio va a tutti gli altri colleghi e amici del NECSTLab, vorrei ringraziarli scrivendo ad uno ad uno tutti i nomi, ma cosí facendo la sezione dei ringraziamenti occuperebbe quasi tutta la tesi.

Un discorso analogo vale per i colleghi universitari con cui ho condiviso momenti fantastici durante questi cinque anni e senza i quali sarebbe stato difficile affrontare lo studio e l'apprendimento con l'umore adatto. Per citarne qualcuno: Daniele Parigi, Luigi Tropiano, Francesco Rotondo, Tommaso Sardelli, Simone Ripamonti e Fulvio Scaramuzza.

Infine dedico questo lavoro ai miei genitori e ai miei familiari, che mi hanno permesso di costruire il percorso di studi che piú mi appassiona, supportandomi e consigliandomi durante tutto questo cammino.

GRAZIE

Abstract

In the last years, the huge amount of available data leads data scientists to look for increasingly powerful systems to process them. Within this context, Field Programmable Gate Arrays (FPGAs) are a promising solution to improve performance of the system while keeping low energy consumption. Nevertheless, exploiting FPGAs is very challenging due to the high level of expertise required to program them. A lot of High Level Synthesis tools have been produced to help programmers during the flow of acceleration of their algorithms through the hardware architecture. However, these tools often use languages considered low level from the point of view of data scientists and are still much too difficult to use for software developers. This complexity limits FPGAs usage in a number of fields, from Data Science to Signal Processing. One way to overcome this problem is to realize Hardware Libraries of widely used algorithms that transparently offload the computation to the FPGA device from high level languages commonly used by data scientists. This work presents different methodologies to create Hardware Libraries for Desktop and Embedded systems. We have chosen to focus on R, MATLAB and Python languages. For what concerns MATLAB and R, the hardware libraries are developed for Desktop systems by the Reusable Integration Framework for FPGA Accelerators to send and receive data to the FPGA connected via PCI-Express. We have implemented and tested an optimized hardware implementation of the Autocorrelation Function on a Xilinx VC707 board and we reached a speedup of 7x with respect to the execution on an Intel i7-4710HQ. Python, instead, is exploited by using the recently released Xilinx PYNQ platform to create Hardware Libraries for Embedded systems. We have implemented different optimized versions of some NumPy library functions for the PYNQ-Z1 Board, that support the PINQ platform. We are able to achieve a speedup of 3.95x for the Integer Matrices Dot Product algorithm implementation and a speedup of 10x for the Correlation function.

Sommario

Negli ultimi anni, la grande quantita' di dati disponibili ha portato gli scienziati di dati a cercare sistemi di calcolo che possano elaborare grandi moli di informazioni ad una velocita' sempre maggiore. All'interno di questo contesto, una soluzione promettente e' rappresentata dalle FPGA, architetture hardware riconfigurabili che consentono di raggiungere elevate performance, pur mantenendo un basso consumo di energia. Tuttavia queste architetture sono difficili da utilizzare a causa dell'alto livello di esperienza richiesto per riuscire ad implementare vesioni hardware ottimizzate degli algoritmi. Per far fronte a questo problema, i maggiori produttori di FPGA hanno sviluppato tool di sintesi ad alto livello, in grado di tradurre codice scritto in linguaggi software di alto livello, in codice che descrive i componenti hardware da configurare sulla FPGA. Ma anche questi tool mirano ad aiutare esperti di design hardware, durante il processo implementativo, e risultano quindi difficilmente utilizzabili da chi non ha conoscenze specifiche del settore, come scienziati di dati o sviluppatori software. Questa barriera d'ingresso sta limitando l'uso delle FPGA in settori dove potrebbero portare enormi vantaggi, come il settore della scienza dei dati e del calcolo scientifico. Un modo per risolvere questo problema e' quello di implementare Librerie Hardware di funzioni largamente usate in questi settori che consentano di utilizzare la FPGA in maniera trasparente all'utente finale. Un Libreria Hardware e', infatti, formata non solo dal codice necessario per configurare la board con la funzione desiderata, ma anche dall'infrastruttura necessaria per eseguire la funzione direttamente dalle applicazioni degli utenti finali, scritte in linguaggi di programmazione a piu' alto livello, quindi facilmente utilizzabili da scienziati di dati e sviluppatori di software. Il nostro lavoro propone diverse metodologie per implementare Librerie Hardware sia per Sistemi Desktop che per Sistemi Integrati. Abbiamo deciso di focalizzarci sui linguaggi a piu' alto livello R, MATLAB e Python, per la loro diffusione nei settori della scienza dei dati e del calcolo scientifico. In particolare, per quanto riguarda R e MATLAB, l'interfaccia e l'implementazione sono state sviluppate per un sistema Desktop, sfruttando lo strumento chiamato RIFFA che consente di comunicare dalla CPU alla FPGA, e viceversa, attraverso un cavo di collegamento PCI-Express. Come caso di studio abbiamo implementato e testato una versione ottimizzata della funzione di Auto-

correlazione presente in una delle librerie standard del linguaggio R. Grazie alle tecniche di parallelizzazione utilizzate siamo stati in grado di ottenere un tempo di esecuzione della funzione, eseguita sulla board Xilinx VC707, 7 volte superiore all'esecuzione software sul processore Intel i7-4710HQ. Per quanto riguarda Python, abbiamo sfruttato la tecnologia PYNQ, recentemente rilasciata da Xilinx, per creare una Libreria Hardware integrata in un sistema Embedded, composta da alcune funzioni della libreria NumPy. Dopo aver testato sia le versioni software che quelle hardware sulla board PYNQ-Z1 abbiamo misurato uno speedup pari a 3.95x, per quanto riguarda l'implementazione del prodotto tra matrici di numeri interi, e uno speedup del 10x per quanto riguarda la funzione di correlazione.
Il resto della tesi e' organizzato come segue:

- il Capitolo 1 dà una visione di insieme del contesto di questo lavoro e brevemente introduce la soluzione proposta;
- il Capitolo 2 fornisce le conoscenze necessarie per capire il lavoro e presenta una visione di lavori collegati, sottolineandone i punti forti e le limitazioni;
- il Capitolo 3 mostra l'implementazione hardware ottimizzata della funzione di Autocorrelazione e l'implementazione della libreria hardware per sistemi Desktop;
- il Capitolo 4 fornisce i dettagli dell'implementazione della Libreria Hardware per il sistema integrato sulla board PYNQ-Z1, la quale supporta la piattaforma PYNQ;
- il Capitolo 5 riporta i risultati dei test effettuati sulle versioni ottimizzate degli algoritmi selezionati;
- il Capitolo 6 discute i risultati e le limitazioni del nostro lavoro, e descrive i possibili lavori futuri.

Contents

List of Tables

List of Figures

Introduction 1

Per sicurezza, dubito di tutto.

Cartesio

This chapter provides an introduction to our thesis. In Section 1.1 we describe the context of this work, while Section 1.2 gives an overview of our proposed solution, and, finally, Section 1.3 outlines the structure of this thesis.

1.1 Context Definition

The interest for data-mining, machine learning and signal analysis has been growing steadily with the need of tools that can process large data, at higher and higher speeds. The natural consequence has been a shift from traditional architectures, to High Performance Computing (HPC) [2] systems with an increasingly high amount of parallelism.

Within this context, Heterogeneous System Architectures (HSAs) [3] are a promising approach to improve performance of the system while keeping low energy consumption. An HSA is composed of different kinds of processing units, like Central Processing Units (CPUs), Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), in a single system to obtain the best resources utilization, in term of execution time and power. CPUs can efficiently run generic tasks, GPUs are optimal for massively parallel repetitive tasks, and FPGAs can be configured to provide a hardware implementation of a set of instructions for an efficient execution. According to TOP500 list [4], one of the most powerful supercomputer in June 2017 is *Tianhe-2*, a system that features both Intel Xeon E5-2692 and Intel Xeon Phi 31S1P co-processor [5], while other supercomputers, like *Titan* and *Piz Daint*, are

accelerated by NVIDIA GPUs. On the other hand, HSAs appear also in June 2017 Green500 list [6], the ranking of the most energy-efficient supercomputers in the world. In fact, the most energy-efficient supercomputer is *TSUBAME3.0*, a heterogeneous system composed of Intel Xeon E5-2680 v4 CPUs [7] and NVIDIA Tesla P100 SXM2 GPUs [8].

However, exploiting these architectures to accelerate applications is a long and hard process that requires time to learn appropriate techniques to get advantages from the hardware and to prototype and test different implementations. Because of these reasons many data scientists still use old tools and libraries of sequential algorithms written in easy-to-use programming languages with strong abstractions in the context of data science and signal analysis. In this thesis, we call these languages Higher Level Languages. By using them, the users can call the functions they need simply by importing the needed libraries, without a deep knowledge of software design. Examples of these languages are Python, MATLAB and R. Modern tools aim at bridging the gap between simplicity and performance. They allow users to remain in the comfort of Higher Level Languages while taking advantage of the parallelism of hardware architectures. Nowadays, GPUs are the most popular choice for improving the performance of data analysis. This is due to a combination of multiple factors, such as their availability and the strong investments made by NVIDIA to promote the parallel computing platform CUDA [9]. The result is a robust support to the most widely used libraries in data analysis and computational sciences, and a strong integration with the most used Higher Level Languages. That said, the interest of data scientists towards FPGAs has recently spiked, thanks to the higher performance per watt and flexibility [10] achievable by these architectures with respect to the GPUs. Thanks to they reconfigurable property, FPGAs are also used to realize heterogeneous embedded systems and devices [11]. The main problem is that they require high level of expertise to exploit their architecture in the right way and the available tools are intended only to hardware design experts.

Even though Computer-Aided Design (CAD) tools to target FPGAs are improving, there is still a gap between the solutions provided by these CAD tools and the needs of data scientists. Over the last years, in fact, High Level Synthesis (HLS) tools allowed to rapidly develop IP

cores and accelerators starting from C code or other languages instead of writing pure Hardware Description Language (HDL) code; however the result of this process still need to be integrated at system level and a set of libraries needs to be written to be ready to use by data scientists.

Nonetheless, ready-to-use libraries and algorithms commonly used in data science that exploit hardware architectures different from CPUs are still very scarce in number. Our work provides the basis for the solution of this problem.

1.2 Thesis Goal

As mentioned in previous Section, the aim of our work is to allow data scientist and software developer to exploit FPGA architectures transparently from their applications written in Higher Level Languages, without approaching the whole hardware design learning path. It is possible to do that by creating Hardware Libraries of widely used algorithms. An Hardware Library is composed of the hardware implementations of different algorithms, by the interface to allow the communication between the CPU, where the user application runs, and the connected FPGA, and, finally, by the integration in one or more Higher Level Languages to call hardware functions transparently. In this way end, users can simply import the Hardware Libraries in their applications, call the functions they need and obtain results faster.

The first step to achieve this goal is to implement a hardware version of one or more algorithms present in the Higher Level Languages software libraries. Then, it is necessary to apply different hardware design techniques to optimize the hardware functions and achieve better performance with respect to the CPU implementation. While implementing the hardware version of the functions, it is also necessary to implement also the communication infrastructure to send data between the CPU and the connected FPGA. This depends also on the type of the target system; in particular, we propose different methodologies for Desktop and Embedded systems. Finally, the Higher Level Languages integration is implemented by exploiting some conversion data type and external function call software libraries.

All these steps are presented in this thesis by using case studies from data science and scientific computing fields.

1.3 Thesis Organization

The work presented in our thesis is organized as follows:

- Chapter 2 explains the necessary background knowledge to understand this work and presents an overview of the related work, underlying the limitations and describing the proposed solution;
- Chapter 3 shows the hardware implementation of the Autocorrelation Function (ACF) algorithm and the design of the hardware library for Desktop systems by exploiting the Reusable Integration Framework for FPGA Accelerators (RIFFA);
- Chapter 4 describes the design of the hardware library for PYNQ-Z1 Embedded system by accelerating software functions from Python libraries;
- Chapter 5 evaluates the results of proposed designs and their integration by comparing software and hardware implementations in terms of execution time;
- Chapter 6 presents a general overview of the results of this thesis, analyzes the limitations of our work, and describes possible future work.

Background and Motivation 2

I believe a leaf of grass is no less than the journey work of the stars

Walt Whitman, Leaves of Grass

This chapter exposes the background of this thesis, and reviews the main tools we are going to employ for such purpose. The chapter, at first, presents an overview of hardware acceleration and the tools we have used to create the hardware libraries for Desktop and Embedded systems (Section 2.1), then analyzes the Higher Level Languages and related tools by underlying their strengths and limitations (Section 2.2). Section 2.3 remarks the motivation of our work and describes our proposed solution.

2.1 Accelerating Algorithms in Hardware

In the computer and electronics world, algorithms can be implemented and computed both in hardware and software with different pros and cons. The main advantage of hardware implementation, such as Application Specific Integrated Circuits (ASICs), is the achievable high performance, thanks to the highly optimized resources to compute specific critical tasks. However, ASICs are permanently configured to only one application and a new fabrication process is necessary to obtain a new hardware implementation that can compute a different algorithm. The software solution, on the other hand, provides the flexibility to change applications and perform a huge number of different task [12], but the performance is orders of magnitude worse than the one achievable with the hardware implementation. A trade off between hardware and software could be obtained by using programmable hardware, such as GPUs

for general purpose computing or FPGAs as reconfigurable devices. In our work when we refer to accelerating algorithms in hardware, we mean porting algorithms on GPUs or FPGAs by applying optimization techniques to achieve a better execution time.
Accelerating algorithms in hardware is often convenient if the program has huge computations. In other cases it is more likely that the program may run slower than it does on the CPU. This is because there are many factors involved to be considered while porting application on hardware [13]. The first one is memory transfer. The data needs to be copied from CPU to the hardware device, then optimized computation is performed and finally the output is transferred back to CPU. This two-way memory transfer between CPU and hardware device is one of the important factors in optimization. As each memory access in the hardware device takes several clock cycles, it is necessary to reduce as much as possible the number of these accesses from the optimized code. If possible, memory transfer and computation should be done in parallel to have better performance. Another main aspect is the algorithm re-engineering. As an example, to exploit the hardware parallelism the problem must be divisible into smaller identical units that can be executed independently each from another. This re-engineering requires a very different approach to traditional problem solving in CPU programming. Moreover, in terms of accelerating algorithms, it is better to know the system architecture in advance and to adapt the solution to it than building a generic solution that can run on more than one architecture: the same algorithm running on two different hardware devices may give very different performance. In particular, the solution has to be designed by keeping in mind: number of cores, memory bandwidth and input data rate, to gain maximum performance.

2.1.1 Graphics Processing Unit

GPUs are, as of today, the most popular choice when it comes to accelerating data analysis. This is due to a combination of factors, such as their availability and the strong investments made by NVIDIA to promote the CUDA project, a platform that enables dramatic increases in computing performance by harnessing the power of the GPUs [9].

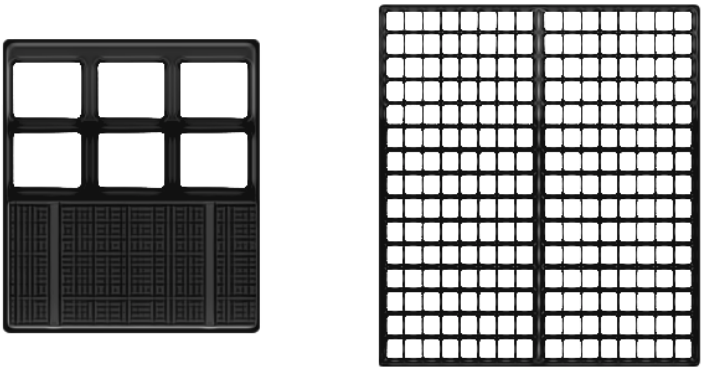

Figure 2.1: CPUs vs GPUs architecture: a CPU consists of a few cores (white squares on the left) optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores (white squares on the right) designed for handling multiple simpler tasks simultaneously [1].

This has led to the development of multiplatform frameworks such as OpenCL [14], with also the support of other companies, such as AMD [15]. The result is a robust support to the most widely used libraries in data analysis and computational sciences, and a strong integration with the most used Higher Level Languages. Thanks to their architecture, GPUs allow to exploit parallelism and gain high performance. Figure 2.1 shows the difference between the number of cores and their dimension between a CPU and a GPU. In particular, a CPU consists of a few cores optimized for sequential serial processing, while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.

In the past, GPUs were intended for graphics computation only. Thanks to the CUDA platform, the modern GPUs are built to be deeply programmable and to support high precision computation. They can be configured by a scalable parallel programming model that extends the familiar C/C++ environment and allows heterogeneous serial-parallel computing.

Even if GPUs have thousands of cores that allow to do a huge amount of computation in parallel and higher upper bound of performance over

CPUs [1], there are some limitations. As an example, GPUs can reach better performance over CPUs only with data-level parallelism applications, while they are slower on purely sequential algorithms due to their architecture. For that reason, getting advantages on GPUs often implies give a completely different shape to CPUs well designed algorithms and small changes in code can produce different orders of performance due to architectural constraints [16]. Moreover, a GPU can not automatically save data to disk if the memory is full (this is especially relevant with some specific languages, such as MATLAB, which can automatically store data on disk if needed, e.g. when operating on large matrices). It operates on vectors of integers and floats, but can not use strings, characters or other data structures. Finally, GPUs do not fare well with control flow instructions: it is better to leave the control flow to the CPU and let the GPU handle the mathematical computation (while keeping in mind that there could be bottlenecks related to data transfer). In particular, GPUs should be used with high amounts of tasks that can be executed in parallel, to take advantage of the higher amount of (slower) processing units, and to minimize the delay caused by data transfer.

2.1.2 Field Programmable Gate Array

FPGAs are field programmable semiconductor devices that can implement arbitrary logic at hardware level, defined by the developer: it is possible to define the operations, the functions and the interconnections to maximize the throughput of a specific task, which is often computationally intensive and inadequate to be carried out on different architectures. In particular, they can be used to efficiently perform computations adapting their programmable structure to a specific algorithm or to a specific part of that [17]. Often, the computationally intensive parts of a program are offloaded to an FPGA, leaving the CPU with less-demanding and serialized code. Figure 2.2 shows the main components of the FPGA architecture [18]. An FPGA is composed of a bi-dimensional matrix of elements called Configurable Logic Blocks (CLBs), a programmable interconnect network to connect the logic blocks, some Input/Output Blocks (IOB) at the periphery of the device that act as the interface between the circuit and the external world and other re-

sources (e.g., clock network, BRAM, DSPs, general purpose processors). A CLB can contain a single basic logic element (BLE), or multiple interconnected BLEs grouped together. BLE is a configurable combinational circuit, usually implemented by means of lookup tables (LUT), devices able to store any n-input combinational function. The FPGA architecture can be programmed to provide high levels of parallelism by applying the same operations on multiple data at once, or by carrying out different operations on multiple data. In fact, thanks to their reconfigurable property, FPGA can cover all cases of the Flynn's taxonomy [19] depending on how they are programmed. Modern FPGAs

As in GPUs, the number of cores of FPGAs architecture is so high that traditional parallelization techniques are no longer effective, and it is necessary to exploit the parallel structure of algorithms at a finer level. Parallelism in FPGAs is achieved at task/data level, by building many workers that apply given functions to the data, and at instruction level, within each worker. Moreover, it is possible to change the data precision in the computation to achieve better performance with negligible error in the results; indeed, in many scenarios, the use of 64 bits operands is expensive and unnecessary. In FPGAs it is easy to mix instructions that work at different levels of precision, even different to the standard 32/64 bits used in Von Neumann machines; as an example, this technique allowed to achieve a 60x speedup over traditional architectures in the case of Lower Upper factorization [20]. Finally, FPGA cores should be able to communicate through high speed links, and should be designed taking into account their desired functionality (such as general purpose arithmetic and efficient vector product).

The main advantage of FPGAs is that they give the possibility to implement a computational architecture that is tailored to a specific algorithm instead of the fixed architecture of CPUs. Then it is possible to change quickly the FPGAs configuration to do different computations on the same piece of hardware. Moreover, FPGAs can reach higher performance with a lower consumption of power, compared to other hardware used as co-processor [21]. However, the possibility to implement a computational architecture implies the capacity to solve the problem from an hardware point of view. In fact, the communication between the FPGAs and the CPUs can introduce problems in term of bandwidth and

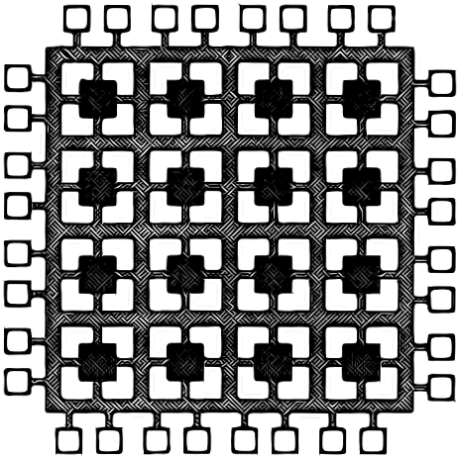

Figure 2.2: FPGAs Generic Architecture: an FPGA is composed of CLBs (black squares), Interconnections (gray lines) and IOBs (external white squares). All the components can be programmed to tailor the hardware architecture to a specific algorithm.

latency that can determine the failure of the attempt of acceleration [17]. Finally, the design time of the optimized algorithm is very long, especially if the developer is not an expert of FPGAs and needs to face with the limited resources of the boards.

2.1.3 Hardware Description Languages and High Level Synthesis Tools

Hardware Description Languages (HDLs) are programming languages used to describe the structure and behavior of digital circuits and can be used to directly program FPGAs. The most popular are VHDL and Verilog and they are officially endorsed IEEE (Institute of Electrical and Electronics Engineers) standards [22]. Most Computer-Aided Design (CAD) tools available in the market support these languages and there are some applications to translate high level languages to HDL, based on some specifications given by the programmer. However writing or generating HDL code to obtain the desired functionalities with good enough performance is a process that requires a hardware design knowledge with specialized skills, and not a simple operation of translating.

For that reason, in the last years, a lot of tools are release to convert high level languages into HDL. These tools are called High Level Synthesis tools and one of them we used within this thesis is Vivado HLS [23]. These tools allow to put some special instructions, in the application code, to generate different hardware architectures and behaviors. However, without a deep knowledge of hardware acceleration, it is very difficult to obtain good results, so only FPGA experts could really exploit them. This limits also the usage of the HDLs.

2.1.4 Reusable Integration Framework for FPGA Accelerators

RIFFA is a framework for communicating data from a host CPU to a connected FPGA via a PCI Express bus [24]. The framework requires a PCIe enabled workstation and a FPGA on a board with a PCIe connector. RIFFA supports Windows and Linux, Altera and Xilinx. The two main functions on software side are data send and data receive. These functions are exposed via user libraries in C/C++, Python, MATLAB, and Java. The driver supports multiple FPGAs (up to 5) per system. On the hardware side, users access an interface with independent transmit and receive signals. The signals provide transaction handshaking and a first word fall through FIFO interface for reading and writing data. No knowledge of bus addresses, buffer sizes, or PCIe packet formats is required. Simply send data on a FIFO interface and receive data on a FIFO interface. RIFFA does not rely on a PCIe Bridge and therefore is not subject to the limitations of a bridge implementation. Instead, RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link.

2.1.5 Xilinx PYNQ platform

For what concern Embedded systems, we have seen a great opportunity in the recently released Xilinx Python productivity for Zynq (PYNQ) platform [25] that allows users to exploit the benefits of FPGAs directly and transparently from their applications written in Python. The board is based on Xilinx ZYNQ technology that integrates a dual-core ARM Cortex-A9 processor (referred to as the Processing System (PS)), with

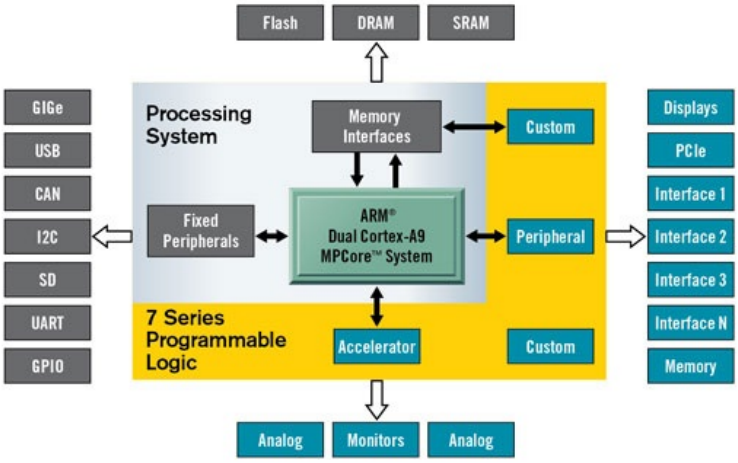

Figure 2.3: ZYNQ Block Diagram

an FPGA fabric (referred to as Programmable Logic (PL)). The *Overlays* paradigm, or hardware libraries, introduced by these platform allows to extend user applications by programming the PL part of the device in a transparent way and to offload part of the computation from the PS part. In this way, it is possible to exploit the FPGA of the board to accelerate the desired application and to provide a software interface written in Python to execute it. Moreover, how it is shown in Figure 2.3, the PL part can directly access to different peripherals, so the user can configure the board to acquire and analyze signals directly on the FPGA, before passing them to the CPU. The figure also shows the main components of the board, and the presence of the Memory interface that allows communication directly from PS to PL part. Thanks to that, it is possible to exploit the communication directly from Python application running on the CPU.

2.2 Higher Level Languages

In the last years, the huge amount of data available and the advent of Machine Learning have allowed the field of Data Science to become more and more popular by impacting on the behavior and the results of a large number of sectors, from Medical Fields to Business Intelligence and An-

alytics [26, 27, 28]. Within this scenario, a lot of Data Science software libraries have been developed for some high level languages to allow the data scientists to use them in an easier way, without a deep knowledge of software design. In our work, we call these programming languages, with strong abstractions in the context of Data Science and Scientific Calculus, Higher Level Languages. By using these languages, the developers can take advantage of the algorithms they need in a transparently way, by simply importing the software library and calling the desired function.

We have chosen to focus on MATLAB [29], Python [30] and R [31], three Higher Level Languages widely used for Statistical Computing and Data Analysis [32]. The following is a list of relevant libraries in the field of Data Science and Scientific Calculus, to offload part of the computation from the CPU to an external device. In fact, these libraries support different hardware architectures to obtain better performance and lower energy consumption than using a general purpose processor. As described in Chapter 1, we are interested in programmable hardware, such as GPUs and FPGAs, thanks to their performance and flexibility.

2.2.1 MATLAB

Despite its decline in popularity in the fields of machine learning and data mining, MATLAB still provides an extremely robust support for GPU acceleration; it features a wide array of GPU-enabled functions [33], ranging from basic linear algebra to specific tasks [34] such as signal processing and bio-informatics. The interaction between MATLAB and the GPU is accomplished by the Parallel Computing Toolbox [35], which is based on the CUDA library. The main building block of the Parallel Computing Toolbox is the `gpuArray`: it is possible to cast any MATLAB array to `gpuArray`, and by doing so the original array will be loaded into the GPU memory, ready to be used. Any instruction that operates on that array will be executed automatically on the GPU, transparently to the user (provided that the instruction is executable by the GPU). Results of the operations executed by the GPU are stored into the GPU memory, but are accessible by the user as if they were saved in main memory (as an example, the plot does not require any

special command to access a `gpuArray`). Results can be brought back to main memory at any time, by using `gather`. `gather` will convert the `gpuArray` as an array of the appropriate datatype (if the original array was made by floats, the new array will also contain floats). It is possible to create objects (i.e. `gpuArrays`) directly on the GPU, without building them as standard arrays and converting them to `gpuArrays`. `arrayfun` can apply a given function to the elements of a `gpuArray`, in parallel. Performance of GPU-accelerated MATLAB code benefits greatly from vectorialized code, which takes advantage of the inherent parallelism of the GPU architecture. It is also possible to write directly [36] CUDA code which is integrated to MATLAB: albeit harder than using ready-to-run functions, writing customized accelerated code can benefit the most advanced applications. MATLAB also support FPGA though HDL Coder [37] and HDL Verifier [38]: these tools can synthesize MATLAB code into HDL, and can be used to port algorithms written in MATLAB to an FPGA. That said, there are currently no MATLAB-FPGA integrations that can rival the Parallel Computing Toolbox in terms of robustness and availability of ready-to-use functions.

2.2.2 Python

Being one of the most widely used programming languages, Python also offers a large number of options in terms of acceleration, spanning both general purpose parallel programming and acceleration of domain-specific tasks. Python can be accelerated via GPU in many ways: two popular general purpose options are PyCUDA [39] and the Anaconda Accelerate [40] package. PyCUDA allows users to call the CUDA API and CUDA C kernels that can be integrated and executed in Python; GPU programming can be kept at a more abstract level by using `gpuArray` [41], which loads data to the GPU and run instructions on them without having to write CUDA code. PyCUDA also offers high-performance linear algebra, by using `gpuArray` in conjunction with NumPy [42], a software library for scientific computing. Moreover, Anaconda Accelerate can automatically accelerate functions by using `@vectorialize` [43], without the need to write GPU-specific code. It also offers bindings to CUDA libraries [44] such as cuBLAS and cuRAND.

In the field of machine learning, a popular Python library is Theano [45], which offers optimized operations on tensors and supports symbolic expressions [46]. The latter is especially useful for machine learning and optimization problems, where it is often needed to compute the gradient of objective functions whose order is dependent on the structure of the data. Theano generates C code for most of its mathematical operations, and then converts them (along with variables) to a graph structure [47]. This is what allows the evaluation and differentiation of symbolic instructions, and the steeper learning curve of Theano is rewarded by higher performance and flexibility.

MyHDL [48] is a hardware description language that can generate Verilog/VHDL from Python code. The idea is to provide to programmers a simple tool to exploit the performance of FPGAs, without having to learn the intricacies of low-level hardware description languages. It should be noted that MyHDL merely converts Python code to Verilog/VHDL, and does not provide any logic synthesis.

2.2.3 R

The language R is widely used for statistical computing and data-analysis, but its out-of-the-box performance is not well suited to deal with the tremendous size of data which is often encountered in these fields. Luckily, there exists a number of ways to accelerate the performance of R and bring it on par with more performing languages. A common approach is to use the Rcpp [49] package, which interfaces R to C++: even without resorting to any parallelization, the improvements over standard R are substantial [50]. Rcpp maps R data types (called `SEXP`) to equivalent C++ classes, but is also able to wrap traditional C++ data types and STL containers to their R equivalent. The high degree of flexibility offered by the package makes it well suited for further accelerations, by using multi-core processors, GPUs or FPGAs. Interfacing R to a GPU can be done by using the gputools [51] package: GPU tools make use of NVIDIA's CUDA language to accelerate numerous data-mining and linear algebra algorithms, such as the distance between vectors, QR decomposition and hierarchical clustering. The popularity of the package is given to its ease of use, as it does not require any knowledge of

CUDA or other languages. Similarly to MATLAB, it is possible to interface R directly to a GPU though CUDA programming [52]. Finally, other packages [53] and tools [54] exist to perform parallel computing by exploiting multi-core CPU systems. As an example, Rmpi [55] is an interface (wrapper) to MPI APIs.

To the best of our knowledge, there are no libraries to connect R applications to FPGAs.

2.3 Problem Definition and Proposed Solution

The need of using alternative solutions to standard CPU and multi-core architectures in the field of data science and signal processing is caused by the availability of unprecedented amount of data to process [56]. Highly parallel architectures, such as GPUs and FPGAs, could be a valid solution to improve the system performance. In fact, both these architectures can be configured to perform Single Instruction Multiple Data computation [57] to accelerate the processing of a large amount of data. Moreover, FPGAs can exploit instruction level parallelism through computational pipelines and, in some cases, hide data transfer latency. The interest of data scientists towards FPGAs has recently spiked, due to the computational power and performance per watt offered by these architectures. The benefits of FPGAs for scientific algorithms have been demonstrated multiple times by works implementing accelerators for different problems. One example is the Data Mining field, where, over the last years, multiple works have proposed FPGA solutions to the implementation of clustering algorithms. As an example, an implementation of a K-Means algorithm has been proposed in [58], while [59] presents a solution for DBSCAN. Some related works about Crosscorrelation Function are proposed in [60, 61], but they use different variants of the algorithm to achieve different results with respect to the one proposed in our work.

Nevertheless, FPGAs are still seen by many as obscure and hard to program, which results in a low amount of tools available for these platforms. As described in Section 2.2 there exists a small number of ways to program FPGAs by using Higher Level Languages, but little to do transparent integration to them, to the contrary of what happens

for GPUs. Often the FPGA-based accelerator is interfaced with the CPU through a host function, written in C/C++, that sends data to the FPGA device and receives back results [62, 63]. This is also the way to interface an FPGA accelerator that exploits the OpenCL standard [64, 65]. The possibility to target FPGA by starting from the OpenCL language [66] is a solution that is gaining traction over the last years, in the context of simplifying the development of hardware cores and the runtime for FPGAs in general. This solution is the one adopted by Xilinx with SDAccel tool [67, 68]. However, as mentioned before, the result is provided to the end user as a C/C++ application difficult to be used by data scientists within their Higher Level Languages applications.

Our solution explores different techniques to solve this problem by creating hardware libraries that allow users to exploit FPGA from Higher Level Languages in a transparent way. In particular, Higher Level Language has the possibility to connect to external libraries and languages by providing specific libraries for converting its internal objects to the ones of the target library or languages. This is the case for MATLAB with the MEX files [69, 70], for Python with CFFI [71] and R with Rcpp [49]. This is the starting point when a Higher Level Language has to be extended to support external libraries and components. It is also possible to connect different Higher Level Languages to the same host function and exploit the build once re-use many times paradigm simply by creating a specific interface for each language to be connected. The second step is to implement the communication between the CPU and the connection FPGA. We also propose different methodologies to do this step, both for Desktop and Embedded systems. In particular, we exploit RIFFA to implement the communication through PCI-Express (PCIe) and the PYNQ platform Overlays concept to implement the communication for embedded devices equipped with ZYNQ technology [72].

Implementation for Desktop Systems 3

See what no one else sees. See what everyone chooses not to see... out of fear, conformity or laziness. See the whole world anew each day!

Patch Adams

This Chapter presents the definition of the Autocorrelation Function algorithm, as well as the description of its software implementation in R, the analysis of its complexity obtained with an initial profiling phase and some consideration of why this algorithm is suitable for being accelerated on FPGA. Then, in Section 3.2, we describe the hardware implementation, the design within a Desktop system by using RIFFA and, finally, the integration with Higher Level Languages.

3.1 Autocorrelation Function

The first algorithm we have decided to accelerate is from the signal processing field: the Autocorrelation Function (ACF) [73]. This algorithm is not only important per se, but it is also a building block of other algorithms such as the Principal Component Analysis or, in general, algorithms that perform dimensionality reduction on datasets by selecting only the most relevant/descriptive features.

3.1.1 Definition

Given two univariate random processes X, Y, with values $x_1, x_2, ..., x_n$, $y_1, y_2, ..., y_n$ over a time-span $1, ..., n$, and defined a *lag* τ, the empirical

(or sample) Correlation Function (CF) $\hat{\rho}_{X,Y}(\tau)$ is defined as:

$$\hat{\rho}_{X,Y}(\tau) = \frac{\sum_{i=1}^{n-\tau}(x_i - \bar{x}_0)(y_{i+\tau} - \bar{y}_\tau)}{\sqrt{\sum_{i=1}^{n-\tau}(x_i - \bar{x}_0)^2}\sqrt{\sum_{i=1}^{n-\tau}(y_{i+\tau} - \bar{y}_\tau)^2}}$$

with

$$\bar{x}_0 = \frac{1}{n-\tau}\sum_{i=1}^{n-\tau} x_i$$

$$\bar{y}_\tau = \frac{1}{n-\tau}\sum_{i=\tau+1}^{n} y_i$$

the sample means of X and Y over interval $n - \tau$. By changing the value of τ, we model the empirical CF of the processes X and Y, which shows the correlation between the processes at various times. The CF shows the degree of similarity of process X with process Y, shifted by a value τ.

If process X is equal to process Y, we get the empirical ACF, that represents the internal similarities of the process with itself. Under the hypothesis of equi-spaced observations one can replace X with Y in the above formula and obtain the following *Autocorrelation* function $r_Y(\tau)$ for process Y with *lag* τ:

$$r_Y(\tau) = \frac{\sum_{i=1}^{n-\tau}(y_i - \bar{y}_0)(y_{i+\tau} - \bar{y}_\tau)}{\sqrt{\sum_{i=1}^{n-\tau}(y_i - \bar{y}_0)^2}\sqrt{\sum_{i=1}^{n-\tau}(y_{i+\tau} - \bar{y}_\tau)^2}}$$

with $y_1, y_2, ..., y_n$ values of Y over a time-span $1, ..., n$, and

$$\bar{y}_0 = \frac{1}{n-\tau}\sum_{i=1}^{n-\tau} y_i$$

$$\bar{y}_\tau = \frac{1}{n-\tau}\sum_{i=\tau+1}^{n} y_i$$

the sample means of Y over interval $n - \tau$. ACF gives information about the randomness of the process and it helps to identify an appropriate time series model (if several *lag* values are analyzed) [74].

3.1.2 R Implementation

The default ACF implementation, available in R libraries as part of package *stats*, processes the input signal into the main routine written in R and then calls a subroutine written in C that computes and returns the final ACF values. The main parameters are a process Y with values $y_1, y_2, ..., y_n$ over a time-span $1, ..., n$ (defined as a univariate or multivariate time series or a numeric vector or a matrix) and the desired maximum lag (lag_max).

The R implementation of the function is composed of two steps. In the first one R creates the input data to be used for the C subroutine. In this step R creates a time series from the input Y and it computes the sample mean $\bar{y}$ of Y as:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

After this, it replaces values of Y with their depolarized value:

$$\hat{y}_i = y_i - \bar{y} \qquad \forall\, i \in [1, n]$$

At this point the C subroutine is invoked.

The C subroutine does the following steps:

1. for each lag $\tau \in [0, lag\_max]$ it computes the corresponding *Sample Autocovariance Function* $\hat{\gamma}(\tau)$ of the input time series (obtained using the normalization by n instead of $n - |\tau|$):

$$\hat{\gamma}_Y(\tau) = \frac{1}{n}\sum_{i=1}^{n-\tau}(y_i - \bar{y})(y_{i+\tau} - \bar{y}) = \frac{1}{n}\sum_{i=1}^{n-\tau}(\hat{y}_i \;\; \hat{y}_{i+\tau})$$

2. for each lag $\tau \in [0, lag\_max]$ it computes the corresponding *Sample ACF* $\hat{r}(\tau)$ of the input time series:

$$\hat{r}_Y(\tau) = \frac{\hat{\gamma}_Y(\tau)}{\hat{\gamma}_Y(0)} = \frac{\sum_{i=1}^{n-\tau}(\hat{y}_i \;\; \hat{y}_{i+\tau})}{\sum_{i=1}^{n}(\hat{y}_i)^2}$$

3. it returns the corresponding ACF $\hat{r}_Y(\tau)$ vector obtained

3.1.3 Profiling

We characterized R implementation of ACF over multivariate signals with increasing number of samples and dimensions, and reported the overall performance of the algorithm, in terms of execution time and quantity of memory allocated/deallocated. To visualize the profiling results we used Profvis, a tool available from GitHub [75]. Profvis samples the state of the function call stack, by stopping the R interpreter at fixed time intervals (by default, every 10ms). Since R sampling profiler results for each execution could be slightly different from one to another [76], we have executed the same profiling test functions multiple times and we have reported the average of the results. Tests have been executed on a Notebook with Intel Core i7-4710HQ CPU (2.50 GHz / 3.50 GHz, 4 core, 6 MB CACHE L3) and 4 GB DDR3L-1600 RAM (3,89 GB usable). To have a meaningful representation of how the performance of the algorithm scale with respect to the size of the dataset, we have changed value of lag_max from the default value: $lag\_max = 10 \cdot \log_{10}(nPts/nDim)$ with $nPts$ being the number of samples and $nDim$ the number of dimensions of the input signals, to: $lag\_max = (nPts/2)$. This is a reasonable hypothesis, if the number of samples is considerable, as in our case, because high ACF values can be found at the extremes of the signal.

The results of our profiling phase is presented in Figure 3.1 and Figure 3.2. Figure 3.1 shows how the execution time (on y-axis) of the ACF, performed on CPU, increase by using univariate signals with a number of points ranging from 50K to 300K (on x-axis). The increase in time complexity is quadratic with respect to the dataset size. Figure 3.2 shows the execution time (on y-axis) of the ACF performed on CPU by using a signal with a fixed amount of points (30K), and an increasing number of dimensions (from 1 to 10) (on x-axis). Once again, the scaling of execution time is quadratic.

By inspecting the source code, we can compute the arithmetic intensity of the algorithm. We considered the number of memory accesses to floating point values, and the number of sums and multiplications performed on floating point values. If n is the number of points of the input signal, p is the number of dimensions, with $N = n \cdot p$, and L is the maxi-

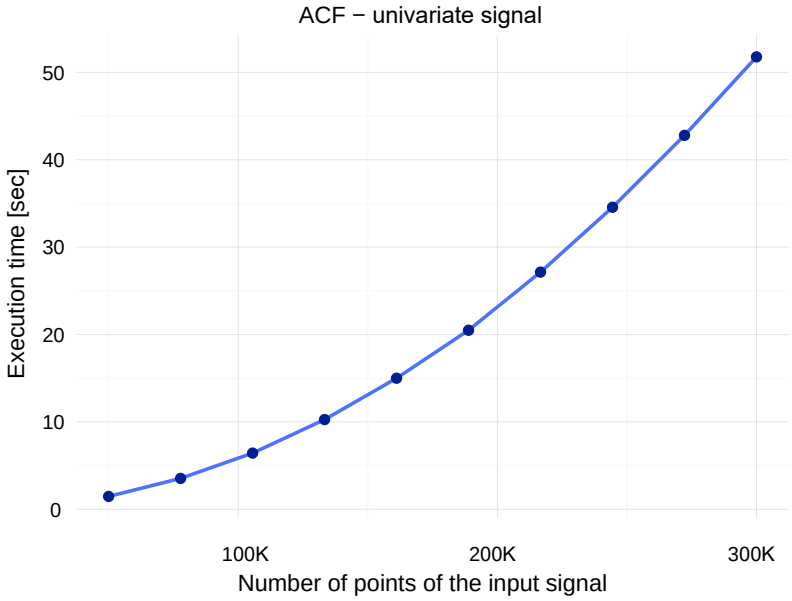

Figure 3.1: Profiling results of ACF on CPU for univariate signals in terms of execution time. The increase in time complexity is quadratic with respect to the dataset size.

mum lag considered in the computation, then the input of the algorithm have approximatively size N and its output has size $p^2 \cdot L$. The number of floating point memory accesses will be about $Lp^2(3+6n) + Lp^2 + p$ which can be asymptotically rewritten as $6Lp^2n$. The number of sums is $Lp^2n + 2Lp^2$ approximatively equal to Lp^2n. The number of multiplications is $Lp^2(1+n) + 2Lp^2$, i.e. Lp^2n. Note that we considered of equivalent complexity multiplications and divisions. p square roots are also performed: their number is, however, negligible compared to the ones of the other operations. As a result, the ratio between floating point operations (FLOPs) and memory accesses is:

$$\frac{FLOPs}{Memory\ accesses} = \frac{Lp^2n + Lp^2n}{6Lp^2n} = \frac{1}{3} = O(1)$$

while the arithmetical intensity, defined as ratio between the number of

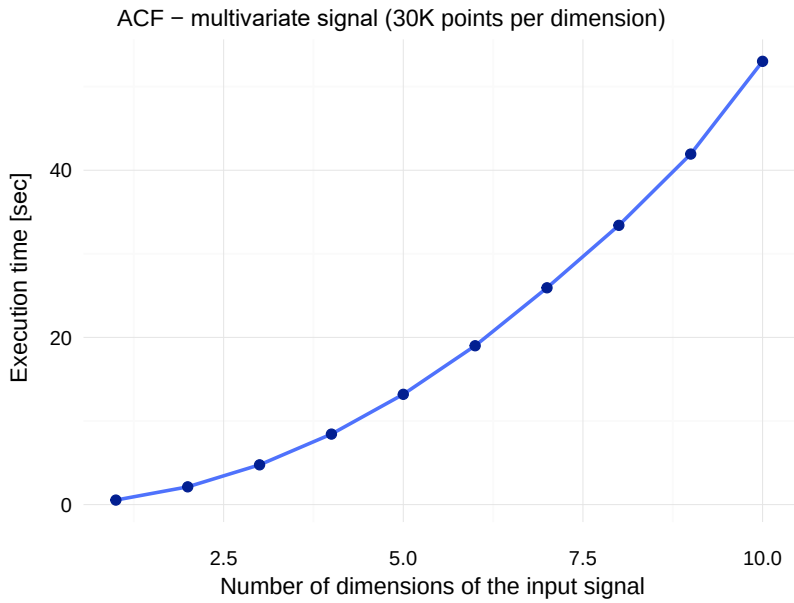

Figure 3.2: Profiling results of ACF on CPU for multivariate signals in terms of execution time. The increase in time complexity is quadratic with respect to the dataset size.

floating point operations and the input size is:

$$\frac{FLOPs}{InputSize} = \frac{Lp^2n + Lp^2n}{np} \approx Lp$$

If the maximum lag L is considerable, with an order similar to the number of points n, the arithmetical intensity becomes:

$$Lp \approx np = N$$

From the results we obtained, it seems possible to improve the performance of ACF in a number of ways:

- *Dimensionality scaling*: in an n-dimensional signal, the computation of the correlation function between two of its dimensions is fully independent of the computation of the correlation functions of other dimensions. As a consequence, it is possible to evaluate in parallel the different correlation functions of the different components of the signal.

- *Lag*: taken the ACF of a univariate signal, the values of this function can also be computed independently from one to another. As an example, the value of the ACF $\hat{r}_Y(\tau = 0)$ can be computed separately from $\hat{r}_Y(\tau = 1)$.

- *Points*: in a given correlation function, for each value of lag, it is necessary to compute many vector products of the time series that compose the signal; the vector product is well suited to be accelerated in a number of ways, such as by making use of pipelined architectures or systolic arrays.

3.2 Proposed Design

The proposed design describe the implementation of ACF acceleration for univariate signals. The solution presented here can be trivially extended to support multivariate signals, but this is not generally useful in practice. To support multivariate signals, it would suffice to re-use multiple times our implementation, with different signals as input. However, we believed more valuable to focus our efforts on optimizing as much as possible the base case of ACF and to put aside these extensions. However, we show in Chapter 4 how it is possible to extend the proposed design of ACF to compute the CF between two different univariate signals.

The first design challenge is that we need to take care on how our core accesses to the input data. In fact, the R implementation has the possibility to access each point of the signal from the host DDR using any stride and type of access with almost no loss in performance due to host cache and pre-caching mechanisms. However on FPGAs random access to DDR on board is a costly operation that can take hundreds of clock cycles. For this reason, to achieve low memory access latencies, it is required to use registers (in the form of LUTs) and BRAM, which, however, are available in limited quantities. It is, in fact, possible to store a reasonable amount of data on the FPGA DDR (from 512MB to a few GB), if needed, but cores can only access BRAMs and registers efficiently which can store data in the order of KB or at most MB.

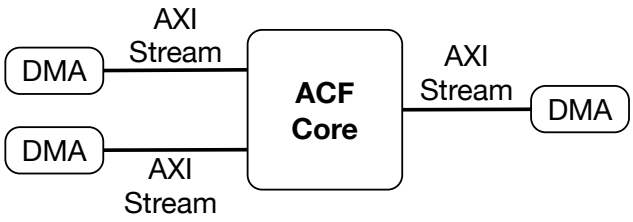

Figure 3.3: Interface of the realized hardware accelerator. The core has two input streams and one output stream. DMAs are used to send and received data between DDR memory and the hardware accelerator.

However, if we look at the data access pattern of our ACF we can see that data are accessed sequentially from memory. Thanks to this, we can design our core to be able to accept data from a streaming interface, so that we do not need to store the whole input signal into local BRAM, but the core can simply access the data from the input streams when new data are needed. A controller is then needed to feed the input streams of the core with the data in the correct order. Being the access sequential, such controller can simply by a DMA which is instructed to copy data from one location to the input stream. Our implementation of the ACF needs 2 input streams, one for the original signal, and one for the lagged one, and an output stream for collecting the output data. This structure defines the interface of our core, as presented in Figure 3.3. The streams are 32 bits wide to accommodate float data which is the datatype used for implementation. Vivado HLS easily allows to define streams and to define the protocol used to communicate with the external component by means of `#pragma` directives. We configured the core to use the standard AXI Stream protocol to communicate with Xilinx DMA cores that will be used to feed data to the core.

After defining the core interface we now describe how we implemented the computation of the ACF. At first, we focused on using as little resources as possible inside the core, by exploiting a stream of input data with an appropriate structure: the idea is that to compute the *i-th* point of the ACF, we need to multiply each point of the signal with the other points, shifted by i. To do so without having to store the entire signal, or all the values of the ACF, inside the core,

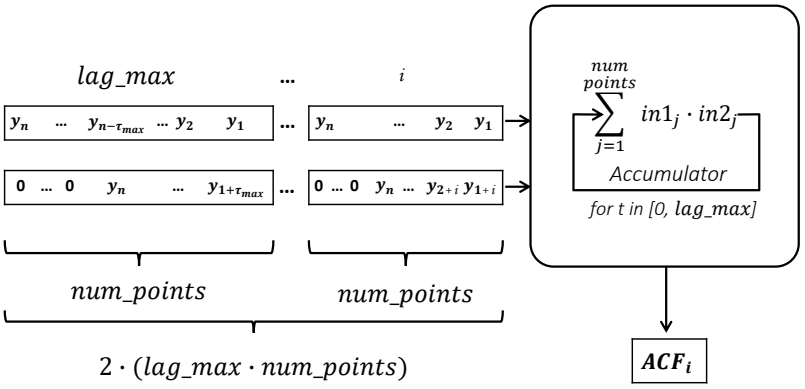

Figure 3.4: Schema of the first FPGA implementation of the ACF. The core has two input streams: one with the signal repeated for each lag to be computed, and the other with the padded shifted signal repeated for each lag to be computed. At each iteration, the core outputs the value of ACF for the lag of that iteration.

we used the 2 input streams in the following way: one contains the signal (with size num_points), and another contains the signal shifted by an amount i. After reading the entire signal, we can output the i^{th} value of the ACF. This process is repeated lag_max times, the desired length of the output. Overall, we need to pass to the cores $(2 \cdot num\_points \cdot lag\_max)$ values, as it is shown in Figure 3.4. By using hardware pipelining, it is possible to mask the cost of expensive multiply-and-accumulate operations, and read from the streams at every clock cycle. This implementation of the algorithm requires about $(num\_points \cdot lag\_max)$ operations, and its time complexity can be approximated to $O(n^2)$, while its spatial complexity is $O(1)$. Unfortunately, having to read $(2 \cdot num\_points \cdot lag\_max)$ becomes extremely expensive even for signals of moderate length: as an example, computing the ACF of a signal of size $50K$ with a lag_max of $25K$, and a clock frequency of $100MHz$, would require:

$$\frac{50K \cdot 25K}{100 \cdot 10^6 \ Hz} = 12.5 \ s$$

while in R this computation would take about 1.5 s.

Moreover, this is an estimation based on the clock cycles required to read data, without taking into account the maximum bandwidth available for the data transfer in the hardware integration, which may slow down execution. It is clear that to achieve high performance it is not enough to take advantage of hardware parallelism, but it is required to lower the amount of data transferred from the main memory. To do so, we decided to add two local buffers inside our core, to store a small portion of the input signal and reduce the input streams size of a factor B (the local buffer size). The idea is that the product of a point x_i and the points in the range $[x_{i+lag},\ ...,\ x_{i+lag+B}]$ will be used to compute the ACF values in the range $[lag,\ ...,\ lag + B]$. To compute the value of ACF at position lag, with lag in the range $[1,\ ...,\ lag\_max]$, we need to multiply and accumulate every point in the original signal and in the signal shifted by a value lag. By using a local buffer that behaves as a shift register, it is possible to compute B values of ACF in parallel; their partial values are stored in the second local buffer, and are written to the output stream only when every pair $\{x_i,\ x_{i+lag}\}$ has been read from the streams. The input streams can be considered divided into blocks: each block has size num_points and allows to compute the ACF values in the range $[1,\ ...,\ lag\_max]$. The number of blocks is equal to lag_max/B. The blocks in the first input stream contain the full signal. The blocks in the second input stream contain the signal shifted by $B \cdot block\_number$. Figure 3.5 shows the delay caused by the initialization of the shift register at the beginning of each block. Once again, by employing hardware pipelining and loop unrolling, we were able to mask the latency of multiply-and-accumulate operations. The number of operations done by the algorithm is still $num\_points \cdot lag\_max$, approximated to quadratic time complexity $O(n^2)$, but now it is required to have two buffers of size B (which is still a $O(1)$ spatial complexity, in fact, B depends only on the available hardware resources). However, as many operations are done in parallel, the execution time of the algorithm is proportional to the input streams size: the local buffers reduce the stream size by a factor B, which in turn reduces the complexity of the algorithm by the same factor. Our reference board, the Xilinx VC707 [77], supported buffers of floats of size 200, partitioned into blocks of 50 floats each, so that it is possible to access 50 data in parallel realizing a

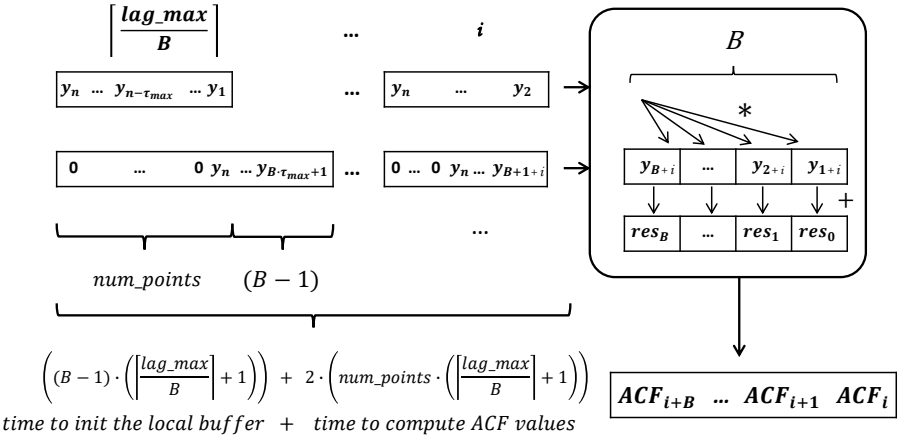

Figure 3.5: Schema of the final FPGA implementation of the ACF. IThe core has two local buffers to store portions of the signal and compute more ACF values in parallel. In this way the length of the two input streams is reduced by a factor equal to the size of local buffers. At each iteration, the core outputs more than one value of the ACF.

SIMD architecture. Being able to use bigger local buffers should lead to even higher performance improvements. Moreover, our implementation can be easily scaled with respect to the available resources on the board, by changing the local buffer size and its partitioning factor, which can be easily done customizing C code given as input to Vivado HLS.

3.3 RIFFA Integration

After realizing the core, we need to perform the system integration phase to realize an hardware architecture which allows us to use the realized accelerator from the host system. The solution proposed in this section is a first prototype of a system that can be used from a host device to perform the computation. The features that have to be made available by the hardware architectures are: (i) the possibility to exchange data via PCIe connection, (ii) the possibility to store data on DDR on board, (iii) the possibility to easily manage the allocation into the DDR memory, and (iv) the ability to control the hardware accelerator in the design. Figure 3.6 illustrates the system we realized for satisfying the

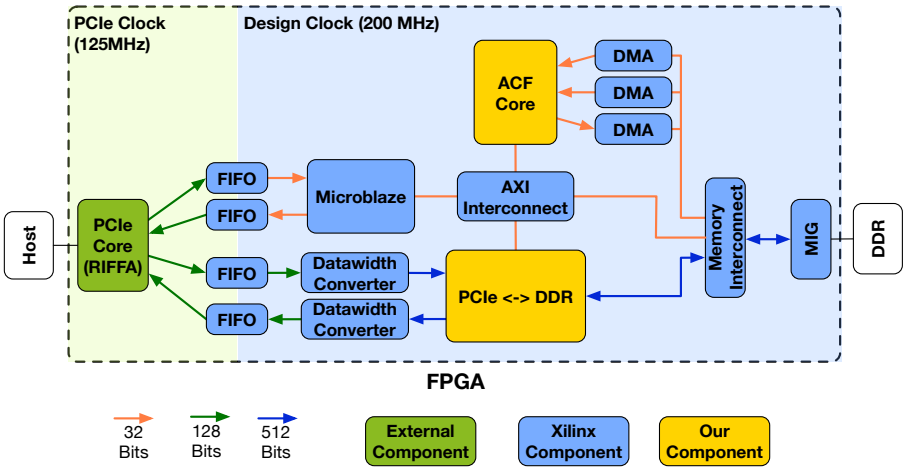

Figure 3.6: hardware Infrastructure to support communication with host.

four points just mentioned.

The PCIe interface has been managed by using the RIFFA, an open source solution from UCSD [24]. This solution, which is made available as pure Verilog, has been packaged in a Xilinx core and extended with the possibility to expose standard AXI4-Stream interfaces instead of the proprietary interface exposed by the original RIFFA core. The part of the design enclosing PCIe runs at 125MHz taking the clock from the PCIe slot, while the remaining of the system runs at the reference frequency generated by the MIG controller, which is the DDR controller made available by Xilinx. This reference frequency can be configured by the user, we used 200MHz in our design. In between these two clock domains we placed FIFO components both because we need buffering to store data coming from PCIe and because the Xilinx Data Stream FIFO component can be configured to act as a clock domain crossing component allowing the synchronization of the two asynchronous clocks present in the design.

The design allows data movement over PCIe to and from the DDR available on the board. Such data movement is possible thanks to a component we wrote in HLS that reads/writes data from/to the AXI Stream connections exposed by RIFFA and communicates with the MIG over

an AXI Master connection. As the transmission is done via PCIe, it is required to have packets of the appropriate size (we use 512 bits packets of 16 floats, also to reduce transfer latencies). We create a hardware design, by using Vivado HLS and Vivado, that can unpack data through specific cores and allow the ACF core handles only 32 bits values, that are then packed and sent back as 512 bits packets, on the PCIe. This core, thanks to the AXI Master interface, is able to issue read and write commands to the DDR and perform data movements. The core has also an AXI Slave configuration port where a controller can pass parameters such as the number of 512 bits words to move and the initial memory address. Upon receiving the start command the cores perform a sequential read or write from the provided initial memory address for the desired number of words. The inputs and outputs of RIFFA communication hardware components are 512 bits width since this datawidth allows to reach a high memory bandwidth; we measured a peak bandwidth of 11.7 GB/s on our Xilinx VC707 board.

The last two requirements (i.e. the possibility to easily perform memory management inside the FPGA and the possibility to control the hardware core) are possible thanks to a Microblaze instantiated in the design. The Microblaze communicates with the host system via PCIe, in fact, one AXI Stream coming from the RIFFA core is directly connected to the Microblaze stream interface. The Microblaze can then receive 128 bits instructions (in 4 32 bit words) from the host. Among these instructions, there is the request to allocate and deallocate a given number of bytes on DDR on-board (MALLOC and FREE). Upon receiving the command the Microblaze performs a *malloc()* or *free()* call and returns the result of the operation to the host, which now is aware of which part of the DDR is ready to be used for the computation. After allocating memory regions inside the DDR the host can copy data to/from the device by sending the Microblaze the proper instruction (DDR_WRITE or DDR_READ) and communicating the initial address and the number of bytes. The Microblaze takes care of configuring the core we implemented for moving data across PCIe to perform the requested operation. At this point, the host can check when the transfer is done with another instruction (CHECK_TRANSFER_DONE). Finally, the host can con-

trol the ACF core by issuing the RUN_CORE command and passing the needed parameters.

3.4 R and MATLAB Integration

The goal of building an interface between Higher Level Languages, such as R and MATLAB, and an FPGA is to provide the user with algorithms that can be called like any other function. These algorithms will transparently make use of an FPGA implementation if certain criteria are met. As an example, when working with data of small size it might not be worth to use an FPGA, as the communication overheads would nullify any gain achieved by using hardware acceleration; in this case, the algorithm would fall back to a traditional CPU implementation. The user would still have to configure its FPGA appropriately (in terms of programming it with the right core), but no specific knowledge of the board hardware and interfaces is required to use the algorithms from Higher Level Languages.

RIFFA, on the software side, exposes two main functions: data send and data receive, available via user libraries in C/C++, Python, MATLAB and Java. To interface Higher Level Languages unsupported by RIFFA, such as R, it requires an additional step. We have created a host C++ function that can be called with the parameters needed to compute the ACF and an output parameter where the values are stored at the end of the call. When it is invoked, it sends the input data to the connected FPGA through the RIFFA interface, then waits for results and writes them into the output parameter. R is connected to the C++ host function by using the Rcpp package offered by CRAN [49]. Rcpp allows to compile C/C++ code and build functions that can be called from R as if they were regular R functions. As R uses its own data types (e.g. arrays of floats use the class NumericVector) we have to convert the passed variable to standard C++ arrays before passing them to the FPGA, and vice-versa when returning the results to R. The conversion is handled by the R's C interface [78] which allows to cast the subtypes of defined SEXP data type to default C++ data types or R data types.

MATLAB is supported by the RIFFA interface and it is possible to use the functions to send and receive data directly from the Higher Level

Language applications. As the FPGA implementation expects the data with a certain structure (depending on the different hardware optimizations), it is not possible to send the signal directly, but a host function that transforms the input signal is needed. To showcase the possibility to connect the same host function written in C++ with more than one Higher Level Language, we have decided to use the one produced for the R application. To connect it to MATLAB we have used the Mex-Files [69]. Similarly to the Rcpp package used by R, Mex-Files allows to call an external C/C++ program from the MATLAB command line as if it was a built-in function. Also in his case, it is necessary to convert the Higher Level Language data types to default C++ data types. MathWorks MEX Library API [70] allows to do the conversion easily.

We also exploit named pipes [79], a method used to send data between different processes (named pipes are created in Linux by using the command `mkfifo()`). The named pipes behave like queues and allow the host application written in C/C++ to run in background and communicate through the queue system the by reading data sent independently from R or MATLAB at runtime.

Finally, even if Boost-Python [80] allows to easily move between Python and C++, as the previous examples, we decided to focus on Python in the Chapter 4 where the recently release Xilinx PYNQ platform is explored.

Implementation for Embedded Systems 4

They say when you meet the love of your life, time stops, and that's true. What they don't tell you is that when it starts again, it moves extra fast to catch up.

Big Fish

This Chapter presents the hardware library of optimized version of some NumPy functions. After a short overview, the details about the implementation are given in Section 4.2, Section 4.3, Section 4.4, Section 4.5. Finally, some properties of our implementation are proposed in last sections of this chapter.

4.1 Overview

This Chapter is focused on extending the PYNQ *Overlays* offer by implementing a hardware library that includes accelerated versions of the core functions of NumPy. We have identified three target functions: the Integer Sum, the Matrix Dot Product and the Correlate functions and for each of them we have implemented both the hardware bitstream to configure the Programmable Logic (PL) of the board and the software APIs to execute and communicate directly from the Processing System (PS). The main idea is to propose a methodology to fully integrate a Python library with the PYNQ platform. In this way the users can exploit the FPGA device in a transparent way, by importing the hardware library and using it instead of the software one. Similar to the ACF case, the system selects at runtime which part of the application can take advantage of the hardware implementation, for example by analyzing the dimension of the input data of a specific function, and executed it on FPGA. If the requirements no fit, the function is executed on the

board processor by calling the original implementation from the software library. With this approach, it is possible to obtain the shortest execution time of the applications by targeting different computing units within the heterogeneous system.

The reasons that led us to choose the Integer Sum, the Dot Product for Integer Matrices and the Correlation functions have been:

- their widespread use within different Data Science fields;
- the possibility to compute independently more than one result value, that can result in exploiting hardware parallelism;
- the capability to demonstrate the applicability of different techniques in integrating functions with the PYNQ platform, such as AXI4-Lite and AXI4-Stream interfaces.

For these purposes, we have used Vivado Design Suite tools by targeting the PYNQ-Z1 board (ZYNQ XC7Z020CLG400-1 part with PYNQ Preset). For each function, the work has been divided into two macro tasks: the Hardware implementation and optimization, and the Software interface creation. Moreover, some of the Software interfaces have been enriched by the optimization of the DMA software routine and Design Reuse was taken into account when designing each overlay. Finally, we have tested each implementation firstly by using Vivado SDK in a standalone application, then by integrating the overlay in the PYNQ platform mounted on the SD card of the PYNQ-Z1 board.

4.2 Integer Sum

The first function we have implemented is the Sum of two integer numbers. We have chosen to implement and describe this very simple function both to show the PYNQ overlay creation process, enriching the examples available to the community, and to demonstrate the possibility to use AXI4-Lite control interfaces and the Python Memory-Mapped I/O (MMIO) library to communicate from the PS to the PL part of the board. By using Vivado HLS, we have created the sum_hw function that exposes three AXI4-Lite ports: two for the operands and another one to control the core and read the result of the operation.

```
int sum_hw(int a, int b){
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=return

  return a + b;
}
```

Figure 4.1 shows the block design of our hardware implementation of the Integer Sum. By enabling one of the General Purpose AXI master interface of the ZYNQ7 Processing System we can connect and directly communicate through the AXI4-Lite with the sum_hw core. The hardware specification produced by Vivado reports the memory addresses needed to communicate with the AXI4-Lite interface.

```
## Base address of sum_hw accelerator
XSUM_HW_BASE_ADDRESS   =  0x43c00000
XSUM_HW_END_ADDRESS    =  0x43cfffff
XSUM_HW_RANGE          =  XSUM_HW_END_ADDRESS -
    XSUM_HW_BASE_ADDRESS

## AXILiteS offsets
## 0x00 : Control signals
##        bit 0  - ap_start (Read/Write/SC)
##        bit 1  - ap_done (Read/COR)
##        bit 2  - ap_idle (Read)
##        bit 3  - ap_ready (Read)
##        bit 7  - auto_restart (Read/Write)
##        others - reserved
## 0x10 : Data signal of ap_return
##        bit 31~0 - ap_return[31:0] (Read)
## 0x18 : Data signal of a
##        bit 31~0 - a[31:0] (Read/Write)
## 0x20 : Data signal of b
##        bit 31~0 - b[31:0] (Read/Write)
```

The MMIO library of Python allows to directly read and write data to a specific memory address. In this way it is possible to control the execution of the core directly from the user application.

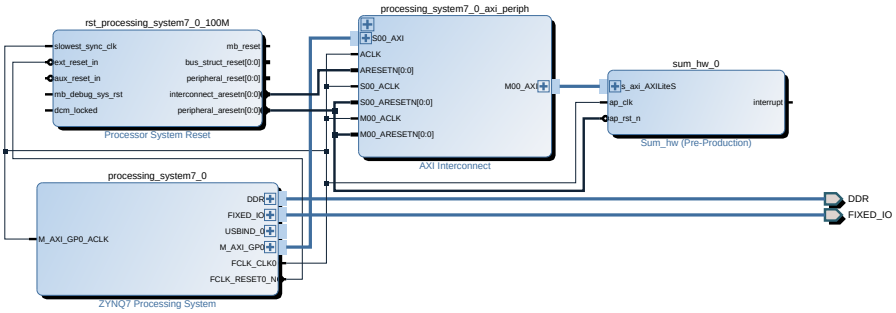

Figure 4.1: Block Design of Integer Sum

```
1  from pynq import mmio
2  mmio = mmio(XSUM_HW_BASE_ADDRESS, XSUM_HW_RANGE)
3  [...]
4  # set the first addend
5  self.mmio.write(XSUM_HW_AXILITES_ADDR_A_DATA, a)
6  # set the second addend
7  self.mmio.write(XSUM_HW_AXILITES_ADDR_B_DATA, b)
8  # start the computation
9  reg_val = self.mmio.read(
       XSUM_HW_AXILITES_ADDR_AP_CTRL) & 0x80 self.mmio.
       write(XSUM_HW_AXILITES_ADDR_AP_CTRL, reg_val | 0
       x01)
10 # wait done signal
11 while((self.mmio.read(XSUM_HW_AXILITES_ADDR_AP_CTRL
       ) >> 1) & 0x1):
12   pass
13 # read result
14 ret_val = self.mmio.read(
       XSUM_HW_AXILITES_ADDR_AP_RETURN)
```

The proposed design of the Integer Sum underlines how it is possible to use simple ports that use the AXI4-Lite light-weight interface to send and receive parameters and control signals between Python applications running on the PS and an core running on the PL. The same approach can be used also in complex systems implementations to influence and control the routines executed on the FPGA.

4.3 Dot Product for Non-fixed Size Matrices

The second function we have implemented is the Matrix Dot Product. The first design challenge is that we need to take care on how our core accesses to the input data. In fact, the software implementation that runs on PS has the possibility to access each point of the matrices from the host DDR using any stride and type of access, as they are stored as arrays in memory. On the other hands, to achieve low memory access latencies from the PL, it is required to use the few available registers. For this reason, it is not possible to store the entire input matrices by only using registers and Block RAMs (BRAMs), unless input matrices have a small number of points. Moreover, we wanted to build an implementation that does not have any fixed size constraint.

Thanks to the fact that to compute the (i, j) point of the output matrix it is possible to access sequentially the points of the i-th row of the first input matrix and of the j-th column of the second input matrix, we have build an core that exploits two input streams of data to compute each point of the output matrix. On the first input stream, we sequentially send the points of the rows of the first input matrix. Each row is repeated for the number of columns of the second input matrix that are sent sequentially on the second input stream. After reading all points of an entire row of the first input matrix and all points of an entire column of the second input matrix (notice that for matrix dot product those number of points are required to be equal), we can write one point of the output matrix on an output stream. Nevertheless, this kind of implementation cannot reach a speedup with respect to software execution by using floating points. This is due to the fact that NumPy uses a highly-optimized, carefully-tuned BLAS method for floating points matrix multiplication, based on the ATLAS project [81]. However, we are able to reach a speedup with respect to the unoptimized NumPy implementation of integer numbers matrix multiplication.

To create stream interfaces we have exploited the hls_stream library and we have mapped the ports with AXI4-Stream interface. By using hardware pipelining optimization, it is possible to mask the cost of expensive multiply-and-accumulate operations, and read from the input streams at every clock cycle. Finally, we have let the system to auto-

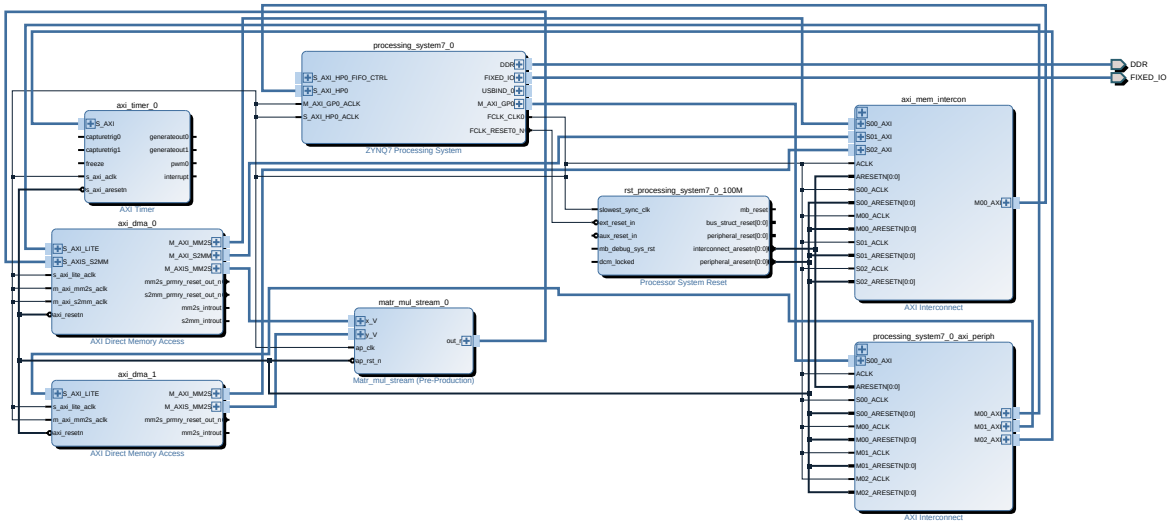

Figure 4.2: Block Design of Dot Product for Non-fixed Size Matrices

matically start the core and we control the execution by sending the desired number of points.

```
#include <hls_stream.h>

struct data_struct{
  int data;
  bool last;
};

void matr_mul_stream(hls::stream<int> &x, hls::
    stream<int> &y, hls::stream<data_struct> &out) {
#pragma HLS INTERFACE axis port=x
#pragma HLS INTERFACE axis port=y
#pragma HLS INTERFACE axis port=out
#pragma HLS INTERFACE ap_ctrl_none port=return

[...]
    for (int i = 1; i < n_iterations *
        n_points_per_iteration; i++) {
#pragma HLS PIPELINE II=1
        accum = accum + x.read() * y.read();
        [...]
    }
}
```

Figure 4.2 shows our block design of Dot Product for Non-fixed Size Matrices. We have inserted two DMAs to send and receive data through the AXI4-Stream interfaces. By enabling one of the AXI high performance slave interface of the ZYNQ7 Processing System we can connect and directly communicate with the DMAs.

Xilinx provides the users with a ready-to-use Python class that manages the DMA operations directly from applications that run on the PS. However, we have decided to reengineer the DMA interfaces to reduce the time overhead introduced by the software APIs. We will describe the proposed optimization in Section 4.6.

Exploiting AXI4-Stream interfaces is a commonly used techniques to send huge amount of data from the host CPU to the connected FPGA. The hardware design of the Dot Product for Non-fixed Size Matrices of Integer Numbers is easily adaptable to different classes of algorithms. In fact, we have used a very similar design, adapted with floating points data streams, in the Correlation function implementation described below.

4.4 Dot Product for Fixed Size Matrices

If we consider small matrices that can be stored by only using registers and BRAMs, it is possible to significantly reduce the number of input points passed to the core, by avoiding duplication of the rows and columns, and to better exploit the hardware levels of parallelism to compute results.

We have fixed the matrix dimension up to size 84x84. We have exploited partition of the local buffer to allow faster access and better parallel execution. The choice of the local buffers dimension and of the partitioning factor depends on the number of hardware resources available. Finally, the pipeline optimization of the nested loops allows the automatic unroll of the multiply-and-accumulate operations.

```
#include <hls_stream.h>

#define DIM 84

struct data_struct{
  float data;
  bool last;
};

void mmult_84(hls::stream<float> &s_in, hls::stream
    <data_struct> &s_out) {
#pragma HLS INTERFACE axis port=s_in
#pragma HLS INTERFACE axis port=s_out
#pragma HLS INTERFACE ap_ctrl_none port=return

    float a[DIM][DIM];
    float b[DIM][DIM];
    float c[DIM][DIM];

    int const FACTOR = DIM/4;
    #pragma HLS array_partition variable=a block
       factor=FACTOR dim=2
    #pragma HLS array_partition variable=b block
       factor=FACTOR dim=1
  [...]
    // matrix multiplication of a A*B matrix
    L1:for (int ia = 0; ia < DIM; ++ia)
        L2:for (int ib = 0; ib < DIM; ++ib)
        {
            #pragma HLS PIPELINE II=1
            float sum = 0;
            L3:for (int id = 0; id < DIM; ++id)
                sum += a[ia][id] * b[id][ib];
            c[ia][ib] = sum;
        }
  [...]
}
```

Figure 4.3 shows our hardware implementation block design of Dot Product for Fixed Size Matrices. We have inserted one DMA that first sends the entire input matrices on the input stream of the core, then, after execution, receives back the entire output matrix on the output stream of the core.

Also in this case, we have used the optimized DMA interface to send and receive data between PS and PL part of the board. One peculiarity is that the design of Dot Product for 84x84 Matrices can be used also

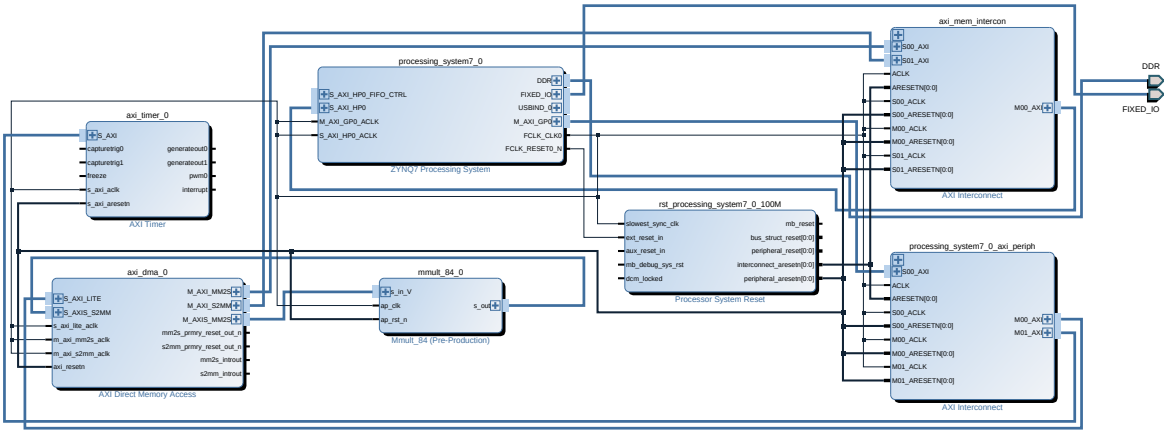

Figure 4.3: Block Design of Dot Product for Fixed Size Matrices

with Matrices of smaller dimensions by padding them wit zeros before the hardware execution.

The design of Dot Product for Fixed Size Matrices underlines how it is possible to exploit the local hardware resources to implement optimization to reduce the execution time and the number of input data to be passed to the core. This is a technique that can be used both when the input data has a small size and when it is possible to divide the input data into blocks to be saved in the local memory of the FPGA and process them separately each from another. If the board provides more hardware resources this design can be easily adapt and reuse to obtain speedup also for matrices of greater dimension.

4.5 Correlation Function

The third function we have implemented is the Correlation function. Like the ACF, It is a function commonly used in signal processing field and in particular it is used to find similarities or repeating patterns in signals. To obtain one value of the Correlation function it is necessary to shift one of the signals of a given lag and then compute the covariance between the shifted signal and the second one. If we repeat this operation among an interval of lags equal to the size of the signals we obtain the

Correlation function. Our implementation contains different types of optimization, trying to combine the techniques exploited both in the hardware implementation of the Dot Product for Non-fixed Size Matrices and the Dot Product for Fixed Size Matrices. In particular the design takes advantage of hardware parallelism both from the pipeline of the operations on the input streams and by exploiting the presence of local buffers to compute in parallel more than one output values.

```
1  #include <hls_stream.h>
2
3  struct data_struct{
4    float data;
5    bool last;
6  };
7
8  int const LOCAL_BUFFER_SIZE = 80;
9  int const PARTITION_FACTOR = 20;
10
11 void correlation(hls::stream<float> &x_in, hls::
       stream<float> &y_lag_in, hls::stream<data_struct
       > &correlation_out) {
12 #pragma HLS INTERFACE axis port=x_in
13 #pragma HLS INTERFACE axis port=y_lag_in
14 #pragma HLS INTERFACE axis port=correlation_out
15 #pragma HLS INTERFACE ap_ctrl_none port=return
16
17   [...]
18     // Use a local buffer of size LOCAL_BUFFER_SIZE
           to calculate the autocorrelation points of
          one iteration.
19     float local_correlation_buffer[
          LOCAL_BUFFER_SIZE];
20 #pragma HLS ARRAY_PARTITION variable=
       local_correlation_buffer block factor=
       PARTITION_FACTOR dim=1
21
22     // Use a shift register to store a certain
          amount of points of the signal.
23     float shift[LOCAL_BUFFER_SIZE];
24 #pragma HLS ARRAY_PARTITION variable=shift block
       factor=PARTITION_FACTOR dim=1
25   [...]
```

```
25    [...]
26        // ============= PARTIAL CORRELATION
              COMPUTATION =============
27      one_iteration: for (int n = 0; n < num_points;
            n++) {
28
29          // Compute a single iteration, i.e
                LOCAL_BUFFER_SIZE values of the
                correlation.
30  #pragma HLS PIPELINE II=1
31          x_i = x_in.read();
32
33          // Shift by 1, and acquire a new point from
                the signal.
34          shift_2: for (int k = LOCAL_BUFFER_SIZE -
                1; k > 0; --k) {
35  #pragma HLS UNROLL
36              shift[k] = shift[k - 1];
37          }
38          shift[0] = y_lag_in.read();
39
40          // ============= PARTIAL SUMS =============
41          // Compute the partial sums.
42          partial_sums: for (int j = 0; j <
                LOCAL_BUFFER_SIZE; j++) {
43  #pragma HLS UNROLL
44              local_correlation_buffer[j] += x_i *
                    shift[LAGS_PER_ITERATION - 1 - j];
45          }
46      }
47    [...]
48  }
```

The design of the Correlation Function is an evolution of the ACF hardware implementation presented in Chapter 3. By using a local buffer that behaves as a shift register, it is possible to compute a number of values of the correlation function in parallel equal to the size of the local buffer; their partial values are stored in the second local buffer, and are written on the output stream only when every points of the signals has been read from the streams. The input streams can be considered divided into blocks: each block has a size equal to the size of the signals. The blocks in the first input stream contains the first signal repeated for an interval of lags and the blocks in the second input stream contains the second signal shifted by the lag contained in the same interval. A

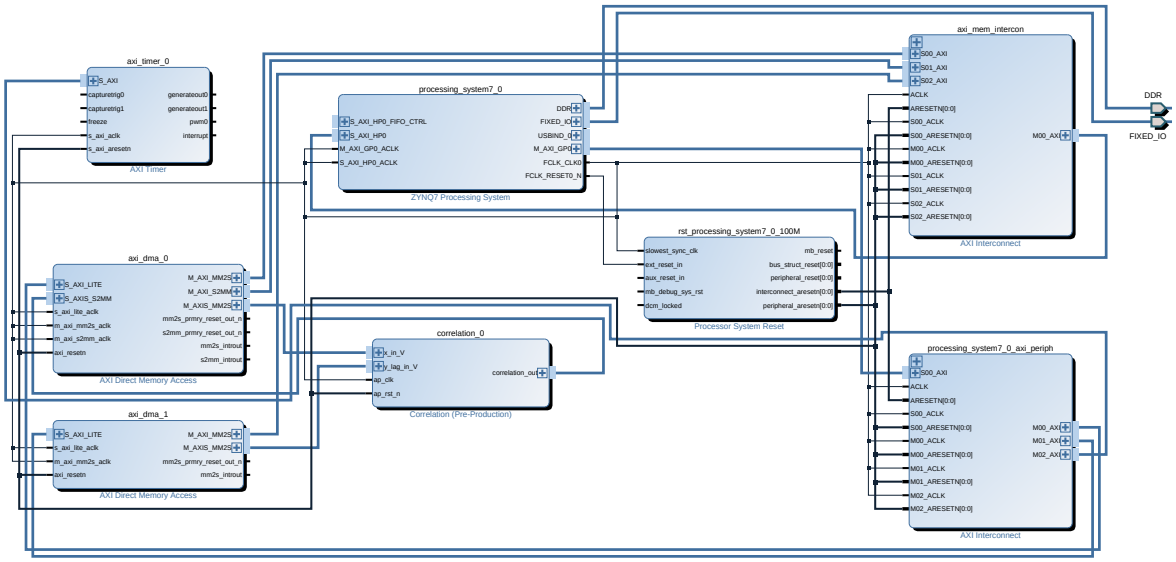

Figure 4.4: Block Design of Correlation Function

padding of zeros, of length equal to the shift, is put in front of each block of the second stream, so that it is possible to read together values from the first and second stream at each clock cycle.

Once again, by employing hardware pipelining and loop unrolling, we were able to mask the latency of multiply-and-accumulate operations. However, as many operations are done in parallel, the execution time of the algorithm is proportional to the input streams size: the local buffers reduce the stream size by a factor equal to local buffers size, which in turn reduces the complexity of the algorithm by the same factor. Being able to use bigger local buffers should lead to even higher performance improvements. Moreover, our implementation can be easily scaled with respect to the available resources on the board, by changing the local buffer size and its partitioning factor.

Figure 4.4 shows the block design of Correlation Function. We have inserted two DMAs to send and receive data through the AXI4-Stream interfaces. By enabling one of the AXI high performance slave interface of the ZYNQ7 Processing System we can connect and directly communicate with the DMAs.

Also in this case, we have used the optimized DMA interface to send and receive data between PS and PL part of the board. The software function also pads the signal before starting hardware execution. This design shows how it is possible to exploit the local hardware resources to implement optimization to reduce the execution time and the number of input data to be passed to the core.

4.6 DMA Optimization

To communicate and control DMAs from the PS, Xilinx provides the users with a ready-to-use Python class that manages the DMA operations. The reasons that led us to decide to reengineer the DMA interfaces are the following:

1. The Python DMA class, provided by Xilinx, is written using a technique called CFFI. CFFI introduces the possibility to call C functions, included in properly compiled C libraries, directly from Python. Although CFFI is considered a good method to improve the Python application performance, in our context, where we need to exclude as much as possible interfacing overheads, another more light and performing solution had to be found.

2. The Xilinx DMA module is intended for a general purpose use and for a potential non-expert audience. For this reason, a series of safety control has been introduced in the code. We can avoid allocating time for some of them since we are in charge of writing the interfaces and the custom DMA module that we are going to provide is not intended for non-expert developers.

3. We needed to approach some of the DMA's routines in other ways. For instance, in the Xilinx DMA class, the initialization of the DMA was coupled with the creation of a local buffer in the board DRAM, while we needed to be able to associate a buffer with a DMA, independently from its initialization.

We chose the Python/C APIs to rewrite our custom DMA module because of their flexibility and expected performance. Python/C API extension module is completely written in C and, thanks to the

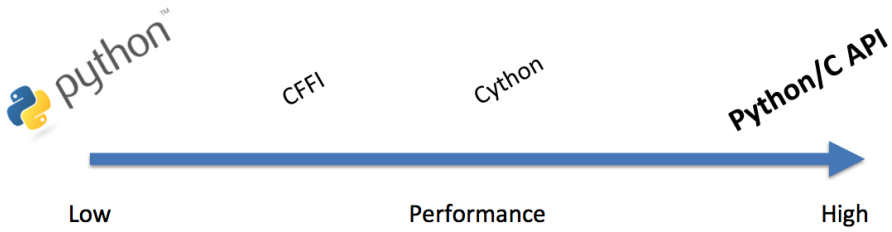

Figure 4.5: Comparison of methods to improve Python performance

Python.h library, a dynamic library can be compiled and then imported into a Python application. The Python object included in the Python.h library assures a fully compatible interaction between the C declarations and the Python code while reducing at the minimum all the overheads between Python and C [82]. Other solutions, like Cython, could have been exploited for our purpose. Although, how it is shown in Figure 4.5, none of them were as straightforward as Python/C APIs and the expected effort needed to get comparable performance was much higher.

In order to extract the maximum speedup possible from the data transmission phase of our applications, we have written custom functions to handle the specific data transmission patterns of each of the functions we accelerated. In this way, the pointers both to the buffers that contain the input data and to the one allocated for the operation result are passed to the specific data transmission function and all the rest of the routine is handled by our C compiled DMA module. In particular, we used some of the XAxi interfaces functions that we also used in the Vivado SDK environment to test our cores.

Thanks to this work, we have obtained an important speedup in terms of the execution time of our hardware accelerated functions ran by the ARM processor mounted on the board. Moreover, we managed to obtain much more flexibility to handle the local buffers in the lightest way.

4.7 PYNQ Integration

Thanks to the PYNQ overlays model, bitstreams can be loaded dynamically by the users directly from their applications and used to configure the fpga. Moreover, thanks to the Python interface included in the overlay, the user will be able to use the hardware accelerated functions just by calling them like functions of a software library. In particular, we have called the produced hardware library, that contains all our optimized function, "numpynq".

From the user point of view, only the Python import will change. If NumPy is imported as follows with Python:

```
1 import numpy as np
2 ...
3 c = np.dot(a,b)
4 ...
```

the numpynq overlay would be loaded as follows:

```
1 import numpynq as np
2 ...
3 c = np.dot(a,b)
4 ...
```

The import of the numPYNQ library will allow the user to use the hardware accelerated functions, as long as they are included in the package, in a transparent way and the overlay will program the board with the bitstream needed to execute the task required by the user. If there is not a hardware implementation of the function called by the user, the numPYNQ library automatically delegates the call to the NumPy software implementation of that function.

Experimental Results 5

L'uomo é un essere fondamentalmente temporale.

Steins;Gate

In this chapter, we show the results that have been obtained, in terms of execution time speedup. These results represent the comparison between our hardware accelerated functions executed by exploiting an FPGA device and the corresponding software implementation of the algorithm executed by the CPU. Finally, Section 5.2.3 describes some software routine techniques to optimize also the integration with Higher Level Languages.

5.1 Desktop System Results

Since the implementation of ACF require PCIe communication components we decided to analyze at first the resource usage of the devised hardware architecture in Section 5.1.1. Then we compare the execution time of the proposed solution with the original R implementation in Section 5.1.2. Our solution has been implemented on a Xilinx VC707 board mounting a Xilinx Virtex7 xc7v456ff157-1 [77] FPGA. The software solution has been tested on an Intel i7-4710HQ [83].

5.1.1 Resource Utilization

Table 5.1 reports a utilization breakdown of the different components in the hardware architecture we designed. Looking at numbers we can see that the ACF core is the one using the most of the resources, but still the remaining of the design, which is used only for handling communication with the host, still occupies a relevant amount of resources. By the synthesis reports, we noticed that the hardware infrastructure, without

Table 5.1: Resource utilization breakdown among different components of the proposed architecture.

Component	LUTs	FFs	BRAMs 18K	DSPs
MIG	13568	15035	3	0
RIFFA	50409	65888	387	0
Microblaze	1749	2103	12	0
FIFOs	860	1523	229	0
Datawidth Converters	401	1564	0	0
Interconnects	8577	10463	160	0
DMAs	2395	3137	15	0
PCIe - DDR	1557	3707	0	0
ACF	83678	113458	0	503

the ACF core, uses: 26% of LUTs, 33% of FFs, and 39% of BRAMs. Most of the resource usage of the infrastructure is caused by RIFFA and the FIFOs used to buffer the data exchanged over PCIe. At the moment this is a limiting factor of our solution since it constrains a number of resources that can be used by the computational core. These numbers are still similar with a number of resources that are used by similar solutions, such as SDAccel [67, 68].

5.1.2 Performance

For evaluating the performance of our solution, we compared the results obtained implementing the ACF algorithm on the Xilinx VC707 board and the corresponding results obtained via native R library. Note that for signals with a low number of points the latency of transfer data through the PCIe can represent a bottleneck, so it results more convenient executing the computation on the CPU. To avoid these marginal cases, we analyzed signals with a number of points greater or equal to $50K$, which is also a reasonable number in the context of data science and signal analysis [84]. The tests were performed using univariate signals with increasing number of points, ranging from $50K$ to $1M$. The maximum lag considered, which coincides with the number of points of the ACF that were computed, is half of the number of points of the signal. From the results in Figure 5.1 it can be immediately seen that

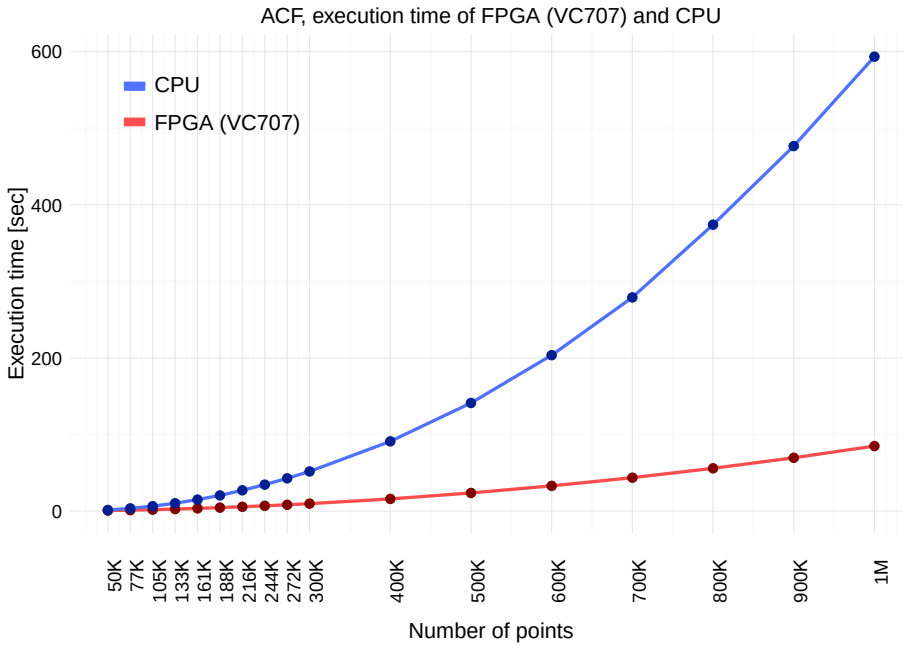

Figure 5.1: Execution time of ACF tests for univariate signal with increasing number of points, on a Virtex-7 and on a CPU. The FPGA massively outperforms the CPU as the signal size becomes bigger and bigger.

the FPGA massively outperforms the CPU as the signal size becomes bigger and bigger. To precisely quantify the improvements of our implementation over the default R one, we can compute the speedup, defined as:

$$Speedup = \frac{Time_{cpu}}{Time_{fpga}}$$

In Figure 5.2 it can be seen how the speedup grows very quickly up to about $300K$ points, then it slows down. This can be explained considering the overheads caused by the data transfer to the core, that becomes negligible for large signals. Moreover, the ability to compute more lags in parallel can only reduce the complexity by a constant factor: this can be seen on large signals, for which the speedup value is almost constant with respect to the size of the input.

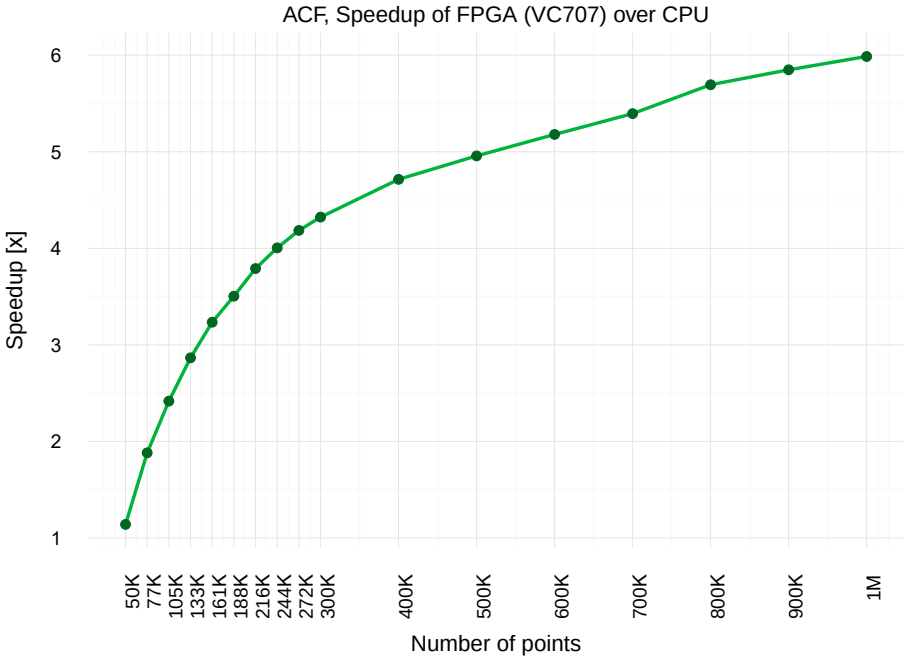

Figure 5.2: Speedup percentage of a Virtex-7 over a CPU for a univariate signal with increasing number of points. The speedup grows very quickly up to about $300K$ points, then it slows down due to the overheads caused by the data transfer and the ability to compute a fixed number of ACF values in parallel.

It should be noted that the speedup curve does not reach a stationary value in our analysis. As we know that the complexity of the considered algorithms is quadratic with respect to the number of points in the input, we performed a polynomial regression (of order 2) over the CPU and FPGA execution times and computed the speedup of the predicted execution times. In Figure 5.3 it can be seen that the speedup stops increasing with signals of more than 5 million points, with a theoretical speedup value of 700%. We also included the real speedup values for signals with $2M$ and $3M$ points, for comparison.

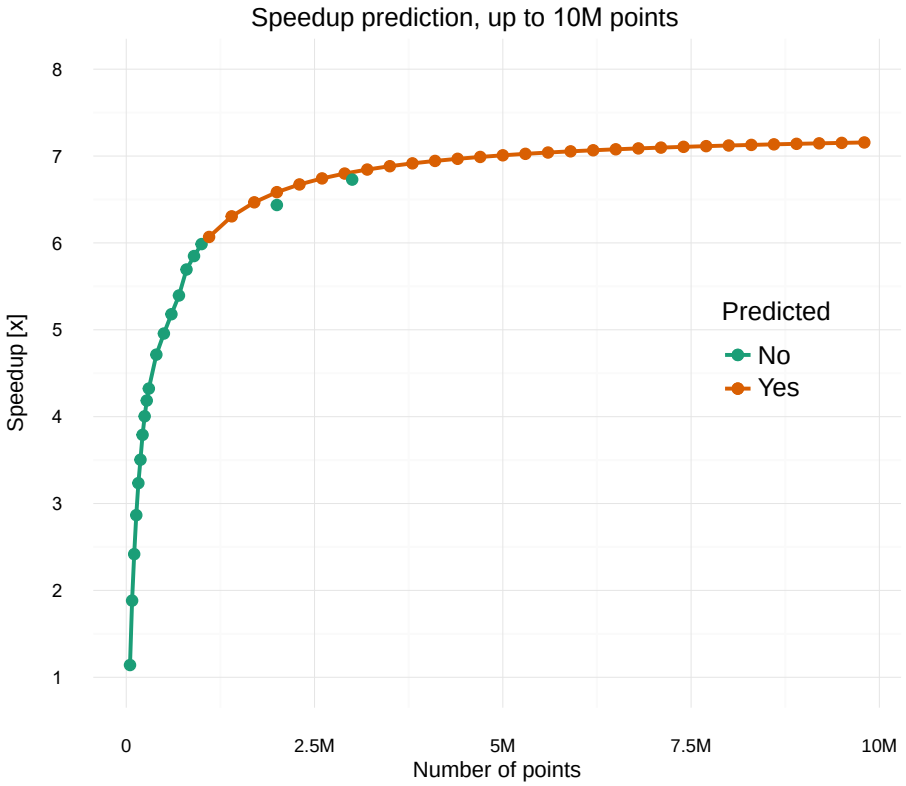

Figure 5.3: Speedup prediction of a Virtex-7 over a CPU for univariate signal with increasing number of points. The speedup curve reaches a stationary value with signals of more than 5 million points, with a theoretical speedup value of 700%.

5.2 Embedded System Results

In this section we compare the execution time of the proposed solution for the PYNQ platform. The results reported are the comparison between the execution times of our hardware accelerated functions and the corresponding NumPy function executed by the dual-core ARM processor mounted on the board PYNQ-Z1. When the optimized version of the algorithm is called by the application, the FPGA is automatically configured with the desired implementation and the computation is offloaded from the processing system of the board to the programmable

logic.

5.2.1 Matrix Dot Product

Both the custom 84x84 dot product and the general one have been tested in two different situations. A test in which the system is highly loaded while another one with the system that is in an idle condition at the moment the test is started. Basically, in the first test type, we launched the execution of dot products of bigger and bigger matrices in which one test suddenly follows the previous one. With the second test type, we let the system go back to an idle condition and then restart the test on matrices of the same dimensions.

For what concerns the custom 84x84 dot product, the test in highly loaded conditions has provided a speedup of 1.09x, while the peak performance of the test in light loaded conditions reached a speedup of 1.35x.

We conducted the two test cases also with the Dot Product for Non-fixed Size Matrices of Integer Numbers. Figure 5.4 shows the tests, executed both for software and hardware implementations of the algorithm, within a loaded system situation in which bigger and bigger dot products have been computed one after the other. For small matrices the CPU has better performance thanks to the data transfer time required by the FPGA implementation. Figure 5.5 reports the speedup curve, that increases with the dimensions of the input matrices. We observed the speedup break even with a dot product between 635x635 matrices .

We conducted the test also in light loaded conditions with a dot product between two 1024*1024 matrices reaching a speedup of 3.95x.

5.2.1.1 Runtime Adaptivity

One of the key result of the Matrix Dot Product implementation, and in general of our work, is the capability of the system to choose the best function implementation based on the input data type and dimension.

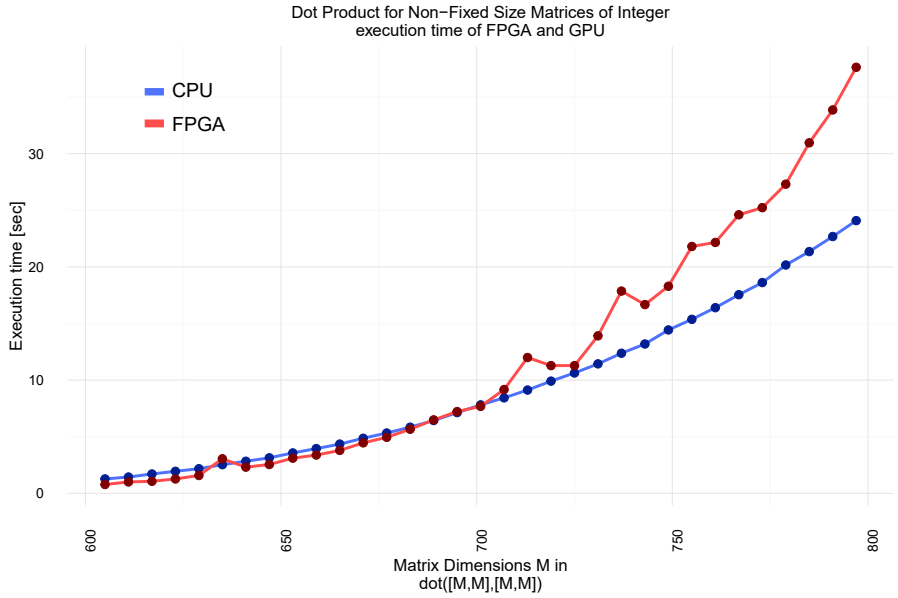

Figure 5.4: Dot Product for Non-fixed Size Matrices of Integer execution time with increasing number of matrices dimensions, on CPU and FPGA of the PYNQ-Z1 board. For small matrices the CPU has better performance thanks to the data transfer time required by the FPGA implementation.

Once a matrix dot product is requested at runtime, our solution decides if the product should be executed by:

- the Dot Product for fixed Size Matrices hardware design made for the matrices with dimensions smaller or equal to 84x84
- the Dot Product for Non-fixed Size Matrices hardware design able to deal with matrices of every dimensions, in particular with size greater than 635x635 (break-even point)
- the NumPy software solution for the other cases

5.2.2 Correlation

We obtained even more interesting results with the test set executed on the Correlation Function. Figure 5.6 shows the execution time of Corre-

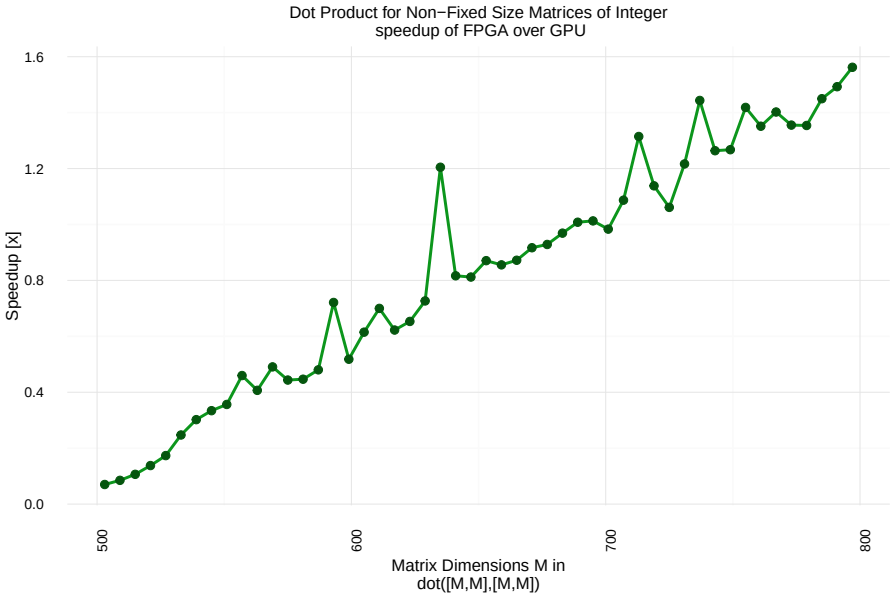

Figure 5.5: Dot Product for Non-fixed Size Matrices of Integer speedup with increasing matrices dimension. We observed the speedup break even with a dot product between 635x635 matrices. This result can be taken into consideration at runtime to select the best implementation to be performed.

lation Function with signals of increasing number of points computed on FPGA and CPU of the PYNQ-Z1 board, while in Figure 5.7 it can be noticed how the speedup has a logarithmic growth related to the number of points contained in the analyzed signals. As it is shown in the graph, the speedup threshold is near 10x. This is due to the fact that the Correlation hardware design implements a series of local buffers in the FPGA memory that allows several optimization in terms of array partitioning and loop unrolling. When the size of the signals becomes very large with respect to the size of the local buffers, no more speedup is expected.

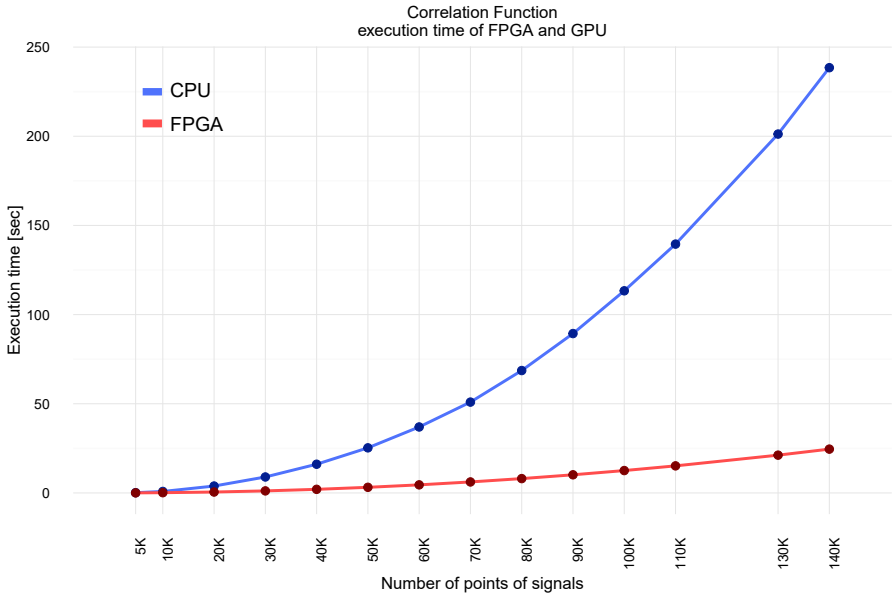

Figure 5.6: Execution time of Correlation Function with signals of increasing number of points computed on FPGA and CPU of the PYNQ-Z1 board.

5.2.3 DMA Optimization

Without an optimized DMA, we experienced extremely slow data transmission that would not have allowed a comparison with the software execution time. We defined the dot product between two 1024x1024 matrices, with our general dot product design, as the test case to evaluate the DMA interface performance. The latest results of our test case resulted in an execution time of 55" with the DMA class provided by Xilinx, while an execution time of 25" with our optimized DMA module. This corresponds to a speedup of more than 2x.

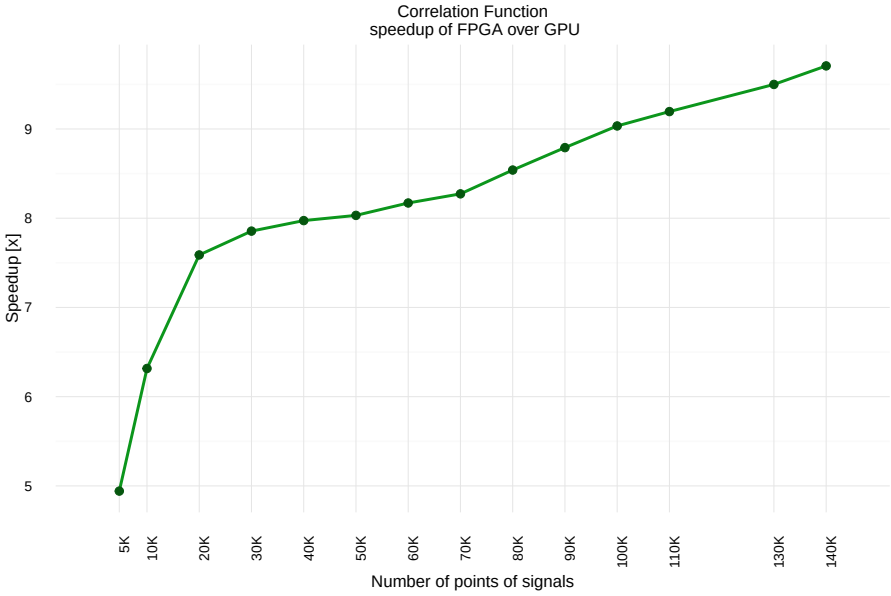

Figure 5.7: Speedup of Correlation Function with signals of increasing number of points.

Conclusions 6

How did it get so late so soon?

Dr. Seuss

This Chapter shows the conclusions derived from our work presented so far. Section 6.1 discusses the contributions of this thesis, while Section 6.2 analyzes the limitations of the proposed work. Finally, Section 6.3 presents some possible future works and enhancements starting from this thesis.

6.1 Contributions

In this work, we firstly presented some techniques to implement FPGA-based optimized version of algorithms that are widely used in the context of data science and scientific calculus. Then, we illustrated how it is possible to take advantage of FPGA-based implementations directly and transparently from Higher Level Languages. To do so, we explored different tools to create hardware libraries both for Desktop and Embedded systems. The main accelerated functions are the software implementations of the ACF present in the default libraries of R language and different implementations of the Matrices Dot Product algorithm and the Correlation Function function from the NumPy Python library.

In particular the main contributions, in the same order as they appeared in the thesis, can be synthesized as follow:

- we have presented a technique to analyze and optimize the ACF algorithm and the design of an FPGA-based hardware implementation to improve its performance, then the methods to integrate the core at System on Chip (SoC) level to allow communication with a Desktop system running Higher Level Languages and the

steps needed to exploit the hardware function transparently from such Languages;

- we have adopted some of the presented techniques to also obtain hardware optimized implementations of Matrix Dot Product algorithm and of the Correlation function from the NumPy library of Python, then we have presented the integration of such functions with the PYNQ-Z1 Embedded system, to offload part of the computation from the CPU to the FPGA. In particular, our work describes how to create Python interfaces to communicate through AXI4-Lite and AXI4-Stream with the custom hardware function configured on the PL;

- we have made the whole exploitation process transparent to the end users so that they can take advantage of the high computational power provided by hardware acceleration without approaching the whole FPGA programming learning path. The system automatically identifies the parts of the user application that can be executed on the FPGA and reduce the overall execution time;

- finally, we have built a software optimized implementation of the Python interfaces that improve the official DMA Class provided by Xilinx within the PYNQ project.

6.2 Limits of the Present Work

The proposed work could be improved with respect to different limitations. First of all, proposed hardware implementations can be refined to support boards with more hardware resources available, for example by leveraging core duplication on FPGA. This could lead to a higher level of parallelism and to better performance, by hypothesizing also an increase of the transfer bandwidth. Secondly, the functions offered to the user are limited to allow the proposed prototypes to be appealing. By using the techniques illustrated in this thesis, it is possible to extend the number of optimized algorithms available to the end user, that can exploit them directly in Higher Level Languages. Similarly, it is possible to add more interfaces to Desktop systems to support new pro-

gramming languages that communicate with the host application. Even with Embedded systems, it is possible to consider the idea of exploring new Higher Level Languages, but this is a more advanced passage that also requires high-level knowledge of the operating system and of the entire architecture of the embedded device. Finally, the illustrated techniques could be generalized and automated to create a framework available for hardware design developers. In this way, they could exploit the interfaces proposed in our work to integrate their core with the supported Higher Level Languages and provide the end user with a ready-to-be-used product.

6.3 Future Work

Future work will focus on multiple aspects to face the described limitation of present work. One will be the support of a wider range of algorithms and the development of the corresponding accelerators and the corresponding interfaces for Higher Level Languages. We will also broaden more support of Hgher Level Languages within the different systems, for instance, a Python interface for Desktop systems. For this purpose, Boost-Python [80] is available to perform the operation to convert datatypes and communicate between Python and the C host application. In doing the extensions in Desktop systems, we might also need to further abstract and revisit the interface with the hardwaredevice to be able to control different hardwarecores via the same interface. Furthermore, we also want to investigate two aspects on the hardwareside, the first one is the support of other PCIe interfaces as the one directly provided by Xilinx in the new versions of Vivado tools or by exploiting SDAccel. Finally, we need to allow the hardwareto perform partial reconfiguration of the core performing the computation to allow the possibility to run different algorithms on the FPGA at the same time.

We also aim to continue to optimize the functions implemented for the Embedded system. First of all, there is a series of optimization that can be included in the hardware library in order to reduce as much as possible the wind up phase that now is needed to set up the FPGA. Then, we think that working on the design of the DMA class, we could reach improvements for what concerns the performance of the data transfer.

To conclude, we would like to adapt our solution, developed on specific boards, to make it work with the much more powerful Xilinx Zynq UltraScale+ [85] or similar boards. These devices have is equipped with a Zynq architecture [72], similar to the one presents in the PYNQ-Z1 board, and for this reason can support the PYNQ platform. Our goal would be to revise the hardware designs, in order to exploit the additional programmable logic that these devices provide, and to let the user interface the hardware accelerated functions in the same transparent way as we did with our first implementation, thanks to the PYNQ Overlays concept.

List of abbreviations

ACF Autocorrelation Function. i, vii, 4, 19–29, 31, 32, 35, 43, 45, 51–53, 61
ASIC Application Specific Integrated Circuit. 5

BRAM Block RAM. 39, 41

CAD Computer-Aided Design. 2, 10
CF Correlation Function. 20, 25
CLB Configurable Logic Block. vi, 8, 10
CPU Central Processing Unit. vi, vii, 1–3, 6–9, 13, 16, 17, 22–24, 41, 52, 62

FPGA Field Programmable Gate Array. i, vi, vii, 1–3, 6, 8–17, 19, 25, 27, 29, 31–33, 35, 38, 41, 43, 51, 53, 55, 61, 62

GPU Graphic Processing Unit. vi, 1, 2, 5–9, 13–17

HDL Hardware Description Language. 3, 10, 11
HLS High Level Synthesis. i, 2
HPC High Performance Computing. 1
HSA Heterogeneous System Architecture. 1, 2

MMIO Memory-Mapped I/O. 36, 37

PCIe PCI-Express. i, 17, 29–31, 51, 52
PL Programmable Logic. 35, 36, 38, 39, 42, 47, 62
PS Processing System. 35, 36, 38, 39, 41, 42, 47
PYNQ Python productivity for Zynq. 11, 17, 62

RIFFA Reusable Integration Framework for FPGA Accelerators. i, 4, 11, 17, 19, 30–32

SoC System on Chip. 61

Bibliography

[1] NVIDIA, "WHAT IS GPU-ACCELERATED COMPUTING?" [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html

[2] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 115–124.

[3] "HSA Fundation." [Online]. Available: http://www.hsafoundation.com

[4] "TOP500 List - June 2017." [Online]. Available: https://www.top500.org/lists/2017/06/

[5] Intel, "Intel Xeon Phi 31S1P coprocessor." [Online]. Available: http://ark.intel.com/products/79539/Intel-Xeon-Phi-Coprocessor-31S1P-8GB-1_100-GHz-57-core

[6] "Green500 List - June 2017." [Online]. Available: https://www.top500.org/green500/lists/2017/06/

[7] Intel, "Processore Intel Xeon E5-2680 v4." [Online]. Available: https://ark.intel.com/it/products/91754/Intel-Xeon-Processor-E5-2680-v4-35M-Cache-2_40-GHz

[8] NVIDIA, "NVIDIA TESLA P100." [Online]. Available: http://www.nvidia.com/object/tesla-p100.html

[9] NVIDIA, "CUDA Parallel Computing Platform." [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[10] F. Ferrandi, M. Novati, M. Morandi, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," in *2006 International Symposium on System-on-Chip*, Nov 2006, pp. 1–4.

[11] V. Rana, M. Santambrogio, and D. Sciuto, "Dynamic reconfigurability in embedded system design," in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 2734–2737.

[12] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation.* Morgan Kaufmann, 2010, vol. 1.

[13] P. Kapadia, "Porting Algorithms on GPU." [Online]. Available: https://www.einfochips.com/blog/partnerships/nvidia/porting-algorithms-on-gpu.html

[14] The Khronos Group Inc., "Opencl." [Online]. Available: https://www.khronos.org/opencl/

[15] AMD, "High-performance computing." [Online]. Available: http://www.amd.com/en-us/products/graphics/server/gpu-compute

[16] D. Barbieri, "Introduzione alla GPGPU." [Online]. Available: http://www.ghostshark.it/cuda/01%20Introduzione.pdf

[17] S. Lab, "FPGA acceleration." [Online]. Available: http://www.synective.se/index.php/acceleration/fpga-acceleration/

[18] A. S.-V. Jonathan Rose, Abbas El Gamal, "Architectures of Field-Programmable Gate Arrays." [Online]. Available: http://isl.stanford.edu/groups/elgamal/abbas_publications/J029.pdf

[19] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[20] J. Gonzalez and R. Nunez, "The art of fpga algorithm design–the case for the extreme acceleration of linear-algebra-intensive software."

[21] ALTERA, "Accelerating High-Performance Computing With FPGAs." [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01029.pdf

[22] M. Ferdjallah, *Introduction to digital systems: modeling, synthesis, and simulation using VHDL*. John Wiley & Sons, 2011.

[23] Xilinx, "Vivado high-level synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[24] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[25] Xilinx, "PYNQ." [Online]. Available: http://www.pynq.io

[26] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact." *MIS quarterly*, vol. 36, no. 4, 2012.

[27] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137–144, 2015.

[28] P. Groves, B. Kayyali, D. Knott, and S. V. Kuiken, "The'big data'revolution in healthcare: Accelerating value and innovation," 2016.

[29] MathWorks, "MATLAB." [Online]. Available: https://www.mathworks.com/products/matlab.html

[30] P. S. Foundation, "Python." [Online]. Available: https://www.python.org

[31] T. R. Foundation, "The R Project for Statistical Computing." [Online]. Available: https://www.r-project.org

[32] G. Piatetsky, "Top Languages for analytics, data mining, data science," 2013. [Online]. Available: http://www.kdnuggets.com/2013/08/languages-for-analytics-data-mining-data-science.html

[33] MathWorks, "Run Built-In Functions on a GPU." [Online]. Available: http://mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html

[34] MathWorks, "Parallel Computing Support in MATLAB and Simulink Products." [Online]. Available: http://mathworks.com/products/parallel-computing/parallel-support.html

[35] MathWorks, "Parallel Computing Toolbox." [Online]. Available: http://mathworks.com/products/parallel-computing/

[36] MathWorks, "Run CUDA or PTX Code on GPU." [Online]. Available: http://mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html#bsic5ih-1

[37] MathWorks, "HDL Coder." [Online]. Available: http://mathworks.com/products/hdl-coder/

[38] MathWorks, "HDL Verifier." [Online]. Available: http://mathworks.com/products/hdl-verifier/

[39] Nvidia, "PyCUDA." [Online]. Available: https://developer.nvidia.com/pycuda

[40] Nvidia, "Anaconda Accelerate." [Online]. Available: https://developer.nvidia.com/anaconda-accelerate

[41] P. documentation, "GPUArray." [Online]. Available: https://documen.tician.de/pycuda/array.html#pycuda.gpuarray.GPUArray

[42] A. Klockner, "PyCUDA: Even Simpler GPU Programming with Python," 2010. [Online]. Available: https://mathema.tician.de/dl/main.pdf

[43] A. DOCS, "CUDA Ufuncs and Generalized Ufuncs." [Online]. Available: https://docs.continuum.io/numbapro/CUDAufunc

[44] A. DOCS, "Anaconda Accelerate." [Online]. Available: https://docs.continuum.io/accelerate/index

[45] deeplearning.net, "Theano." [Online]. Available: http://deeplearning.net/software/theano/

[46] F. Bastien and P. L. et al., "Theano: new features and speed improvements," 2012. [Online]. Available: http://arxiv.org/pdf/1211.5590.pdf

[47] deeplearning.net, "Graph Structures." [Online]. Available: http://deeplearning.net/software/theano/extending/graphstructures.html#graphstructures

[48] MyHDL, "Design hardware with Python." [Online]. Available: http://www.myhdl.org

[49] C. R-project, "Rcpp: Seamless R and C++ Integration." [Online]. Available: https://cran.r-project.org/web/packages/Rcpp/index.html

[50] D. Smith, "A comparison of high-performance computing techniques in R," 2015. [Online]. Available: http://blog.revolutionanalytics.com/2015/06/a-comparison-of-high-performance-computing-techniques-in-r.html

[51] C. R-project, "gputools: A Few GPU Enabled Functions." [Online]. Available: https://cran.r-project.org/web/packages/gputools/index.html

[52] P. Zhao, "Accelerate R Applications with CUDA," 2014. [Online]. Available: https://devblogs.nvidia.com/parallelforall/accelerate-r-applications-cuda/

[53] C. R-project, "CRAN Task View: High-Performance and Parallel Computing with R." [Online]. Available: https://cran.r-project.org/web/views/HighPerformanceComputing.html

[54] R. Tutorial, "GPU Computing with R." [Online]. Available: http://www.r-tutor.com/gpu-computing

[55] T. R. Foundation, "Rmpi." [Online]. Available: https://cran.r-project.org/web/packages/Rmpi/index.html

[56] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[57] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[58] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on.* IEEE, 2011, pp. 475–480.

[59] N. Scicluna and C.-S. Bouganis, "Fpga-based parallel dbscan architecture," in *International Symposium on Applied Reconfigurable Computing.* Springer, 2014, pp. 1–12.

[60] X. Wang and X. Wang, "Fpga based parallel architectures for normalized cross-correlation," in *2009 First International Conference on Information Science and Engineering*, Dec 2009, pp. 225–229.

[61] B. Miao, R. Zane, and D. Maksimovic, "A modified cross-correlation method for system identification of power converters with digital control," in *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, vol. 5, June 2004, pp. 3728–3733 Vol.5.

[62] A. Parashar, M. Adler, M. Pellauer, and J. Emer, "Hybrid cpu/fpga performance models," in *3rd Workshop on Architectural Research Prototyping (WARP 2008)*, 2008.

[63] M. D. Santambrogio, H. Hoffmann, J. Eastep, and A. Agarwal, "Enabling technologies for self-aware adaptive systems," in *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, June 2010, pp. 149–156.

[64] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From

opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[65] Intel, "Intel fpga sdk for opencl." [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf

[66] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[67] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu, "Optimizing opencl applications on xilinx fpga," in *Proceedings of the 4th International Workshop on OpenCL*. ACM, 2016, p. 5.

[68] G. Guidi, E. Reggiani, L. Di Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve fpga-based systems design productivity via sdaccel," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 247–252.

[69] MathWorks, "Introducing mex files." [Online]. Available: https://it.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html

[70] MathWorks, "Mex file creation api." [Online]. Available: https://it.mathworks.com/help/matlab/call-mex-files-1.html

[71] Armin Rigo, Maciej Fijalkowski, "C Foreign Function Interface for Python." [Online]. Available: https://cffi.readthedocs.io/en/latest/

[72] Xilinx, "Zynq-7000 All Programmable SoC." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[73] P. F. Dunn, *Measurement and data analysis for engineering and science*. CRC press, 2014.

[74] NIST/SEMATECH, "e-Handbook of Statistical Methods." [Online]. Available: http://www.itl.nist.gov/div898/handbook/eda/section3/eda35c.htm

[75] W. Chang, "Profvis intro." [Online]. Available: https://rpubs.com/wch/123888

[76] Winston Chang, "Profvis intro." [Online]. Available: https://rpubs.com/wch/123888

[77] Xilinx, "Virtex-7 fpga vc707 evaluation kit." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

[78] Advanced R by Hadley Wickham, "R's c interface." [Online]. Available: http://adv-r.had.co.nz/C-interface.html

[79] The Open Group, "mkfifo - make a fifo special file." [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/mkfifo.html

[80] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with boost. python," *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[81] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–27.

[82] P. S. Foundation, "Python/C API Reference Manual." [Online]. Available: https://docs.python.org/3/c-api/intro.html

[83] Intel, "6th Generation Intel Core i7 Processors." [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html

[84] D. Ruppert, *Statistics and data analysis for financial engineering*. Springer, 2011, vol. 13.

[85] X. Inc., “Zynq UltraScale+,” Available at https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (June 25, 2017).