POLITECNICO DI TORINO

MASTER's Degree in INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

MASTER's Degree Thesis

Hardware Acceleration Based on FPGA for 5G NR Link Layer Simulator

Supervisors

Prof. LUCIANO LAVAGNO

Ing. SALVATORE SCARPINA

Ing. ROBERTO QUASSO

Candidate

ALBERTO CARBONI

MARCH 2020

Sommario

Il lancio di ogni nuova tecnologia mobile viene preceduto da una procedura fondamentale di test, la quale si compone, prima di ogni altra, di importanti fasi di simulazione. Queste si rivelano quanto mai essenziali per diverse ragioni.

Nella pianificazione di rete (dall'inglese *network planning*), è importante stabilire con anticipo le prestazioni che si otterranno dall'attrezzatura impiegata sul campo ancora prima che questa sia fisicamente installata. Ciò permette di determinare la configurazione ottimale che soddisfi ai requisiti dello standard, ad esempio in termini di copertura di rete e latenza.

Sarebbe altresì impensabile installare antenne e più in generale qualsivoglia apparecchiatura senza una precisa stima del risultato. Questo per l'impossibilità di muovere l'equipaggiamento e tracciarne la performance reale con tempistiche e costi adeguati.

Una stima a monte è oggi possibile grazie a complesse modellazioni matematiche come i modelli di canale (*channel models*). Si tratta, in breve, di descrizioni del mezzo tra una o più antenne trasmittenti e una o più antenne riceventi, ricche di parametri variabili funzioni di tempo, frequenza e spazio. Gli effetti più comuni modellati sono, tra i tanti, la perdita di potenza del segnale a causa di caratteristiche ambientali - dalle quali derivano rifrazione e riflessione - e il Doppler.

È dunque facile rendersi conto di quanto gravosa possa essere una simulazione dal punto di vista del calcolo. Il lavoro nasce proprio dall'esigenza di velocizzare questa fase poiché troppo lenta al fine di ottenere risultati soddisfacenti in tempi consoni.

La tesi tratta della valutazione circa l'accelerazione hardware di questa procedura tramite FPGA, hardware riprogrammabile per molteplici occorrenze. Le FPGA possiedono un'ottima capacità di parallelizzazione delle operazioni, la quale permette di ottenere un'accelerazione generalmente tra le 10 e le 100 volte del tempo di esecuzione di un codice su processori tradizionali. I modelli considerati nel seguito sono delle schede di sviluppo costruite attorno ad una FPGA, facilmente integrabili a server (o desktop) tramite porta e annesso protocollo PCIe. Per la programmazione delle schede si è utilizzato il linguaggio OpenCL, standard nei

sistemi di calcolo cosiddetti eterogenei. OpenCL si basa sul linguaggio di programmazione ad alto livello C.

Si è partiti dal codice sorgente fornito da TIM S.p.A., ossia dall'implementazione software della simulazione del modello di canale. Il codice originale è stato analizzato per poi essere ottimizzato sino al massimo delle prestazioni teoriche sulla piattaforma hardware in esame. Infine, è stata tracciata l'esecuzione su piattaforma fisica, in cui sono emersi alcuni aspetti di differenza importanti.

Il processo di perfezionamento non si è rivelato banale; al contrario, il codice è stato sottoposto ad ampie trasformazioni manuali, a causa della scarsa capacità di ottimizzazione automatica del software necessario alla programmazione.

Summary

The launch of every new mobile technology is preceded by a fundamental test procedure, which is composed, before any other, of critical simulation phases. These prove to be essential for many reasons.

In the so-called network planning, it is crucial to establish in advance the performance obtained from the on-field appliance even before it is physically installed. That allows, in general, to determine the optimal configuration that meets the requirements of the standard, e.g. in terms of network coverage and latency.

It would also be unthinkable to install antennas and more generally any equipment without a precise estimate of the result. Principally that comes from the impossibility of moving heavy hardware devices and tracking real performances with adequate timing and costs.

An upstream estimate is nowadays possible thanks to sophisticated mathematical representations such as channel models. These are, in short, descriptions of the medium between one or more transmitting antennas and one or more receiving antennas, with many variable parameters, functions of time, frequency, and space. The most common effects modeled are, among many others, the loss of signal strength due to environmental characteristics - from which refraction and reflection derive -, and the Doppler.

It is, therefore, easy to realize how burdensome a simulation can be from the computation point of view. The work comes precisely from the need to speed up this phase as it is too slow in order to obtain satisfactory results in a suitable time.

The thesis deals with the evaluation of the hardware acceleration via FPGA, reprogrammable logic for various occurrences. FPGAs have an excellent ability to parallelize operations, which allows obtaining an acceleration generally between 10 and 100 times the execution time of code on traditional processors. The hardware considered are development boards built around an FPGA, which can be easily integrated within servers (or desktops) via PCIe. For the programming of the accelerator cards, the OpenCL language was used, standard in heterogeneous computing systems. OpenCL is based on the high-level programming language C.

The work started with the source code provided by TIM S.p.A., i.e., with the software implementation of the channel model simulation. The original code was

analyzed and then optimized to the maximum theoretical performance on the hardware platform in question. Finally, execution on the physical platform was traced, in which some essential difference aspects emerged.

The refinement process was not trivial; on the contrary, the code has undergone extensive manual transformations, due to the limited ability to optimize the software by the software synthesis tools employed.

Table of Contents

List of Tables

List of Figures

Acronyms

3GPP 3rd Generation Partnership Project

AFI Amazon FPGA Image

ALU Arithmetic Logic Unit

AMI Amazon Machine Image

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AWS Amazon Web Services

BRAM Block Random Access Memory

C-RAN Cloud Radio Access Network

CDL Cluster Delay Line

CLB Configurable Logic Block

CMOS Complementary Metal Oxide Semiconductor

CPU Central Processing Unit

D2D Device-to-Device

DDR Double Data Rate

DSP Digital Signal Processor

EC2 (Amazon) Elastic Compute Cloud

ETSI European Telecommunication Standard Institute

FIFO First In First Out

FF Flip-Flop

FPGA Field Programmable Gate Array

gNB (3GPP) Next Generation Node B

GPU Graphics Processing Unit

HDK Hardware Development Kit

HDL Hardware Description Language

HLS High-Level Synthesis

IoT Internet of Things

ITU-T International Telecommunication Union

LOS Line-Of-Sight

LTE Long-Term Evolution

LUT Lookup Table

M2M Machine-to-Machine

M-MIMO Massive Multiple Input Multiple Output

NFV Network Function Virtualization

NLOS Non-Line-Of-Sight

NR New Radio

P2P Peer-to-Peer

QoS Quality of Service

RAM Random Access Memory

RAT Radio Access Technology

RF Radio Frequency

ROM Read Only Memory

RTL Register-Transfer Level

SDK Software Develpment Kit

SCM Space Channel Model

SDN Software-Defined Networking

SLR Super Logic Region

SRAM Static Random Access Memory

TDL Tapped Delay Line

UE User Equipment

XOCC Xilinx OpenCL Compiler

Chapter 1

Introduction

This introductory part will deal with listing the structure of the thesis, together with an exposition of the methodologies followed. As briefly anticipated in the summary, the work arises from the need of TIM S.p.A. (from now on TIM) to speed up a simulation phase. This phase, in particular, is quite burdensome from a computational point of view, despite the successful attempt by the insiders of first optimizing the code for the processors of their data centers.

In recent years, a trend involving the transfer from original hardware to software virtualized functions has been established. This transit allows companies to disengage from expensive and sophisticated network equipment by taking advantage of low-cost conventional ICT products. However, it is simple to realize how the more these virtual functions model low-level tiers (descending the network stack), the more they need computing power, given the increasing details managed. The simulation examined here, being at the link-layer, is an example of how complexity reaches a feasibility limit. This is the reason why integrating a hardware accelerator like an FPGA, tightly coupled with the CPU of an original machine called host, was taken into account.

This thesis has been carried out independently, even though TIM is cooperating with Politecnico di Torino with a broader group. The required task was to speed up the articulated simulation phase as much as possible, starting from the original C++ code. As well explained in the initial section of chapter 4, the first version given us by TIM also included some MATLAB code.

In this work, a previous master's thesis [1] will be mentioned, on another part of the same large simulation of the link layer. The methodology adopted by the author Nasir Ali Shah was taken as a starting point, being the thesis quite similar in the topics covered. However, the two works differed sharply in the data structures and algorithms involved. These differences allowed customizing the work and achieving the final result in order to be able to integrate the two. Some words regarding this integration step will be spent in the final chapter.

As for the other chapters, after this introductory one, the discussion will move on to two other general parts in support of the topic. More in detail, chapter 2 explains the underlying reasons for channel simulation in the launch of new technologies and mobile standards. The starting point will be the technological innovations brought by 5G, which is well known to bring with it a noteworthy and generational change of mentality. It will not be a matter of guaranteeing a new access speed or broader coverage; thanks to these reachings, small revolutions will take place not only in the telecommunication sector but also in the industrial, manufacturing, public, and health services, to cite a few.

The supporting technical innovations are vast and significant, of course, and testing/simulation must necessarily keep up and take into account these new paradigms. Always within chapter 2, the focus will then move on to a brief overview of the types of the most popular channel models, reviewing the most frequently covered aspects.

Chapter 3 aims instead to be closer to the devices on which acceleration is based. These are FPGAs (Field Programmable Gate Arrays), circuits reprogrammable through a bitstream generated by a synthesis tool. It was decided to consider in this work various models by Xilinx, one of the leading companies in the FPGA market. The reference software suite was Vivado SDx (version 2018.2), capable of integrating some host code, running on a desktop or server, with kernel code, running instead on a demanded hardware device. Vivado SDx also manages the interfacing, data migration, and generation of the bitstream to program the FPGA. This software tool takes advantage of high-level synthesis. A few words on this and the advantages compared to the use of hardware description languages will be spent in a subsection of chapter 2. Finally, at the end of it, the OpenCL standard will be mentioned in the context of heterogeneous computing. Also, the models exposed by OpenCL for what concerns code execution and memory hierarchy will be briefly mentioned.

Chapter 4 deals with the actual thesis work. The thesis does not want to present or confirm the use of an experimental scheme. It is a real case of application to be accelerated, and it is a work highly oriented towards a final result. The methodology can be considered partly experimental in the succession of its steps, as it does not derive from any standard or fixed procedure.

The followed method will be emphasized later several times in the text. First of all, the necessary time to become familiar with the development environment has been taken. In the university courses held so far, there was no specific preparation for the tools used here. On the contrary, in some of the teachings, reprogrammable architectures were studied, and FPGAs programmed using hardware description languages such as VHDL.

As a second step, the TIM code was studied, and its operation analyzed first in outline, then in detail. This extensive understanding was forced by the need

to know, for example, the data access pattern, in order to make a significant improvement when modifying the code. The code itself has been modified not only to adapt to the new hybrid architecture but to take full advantage of the hardware platform used. The work was not entirely smooth nor quick, given on the one hand an initial period of training, and on the other the complexity of the code. During an initial training period, some pre-installed examples in the software tool were seen. Then, an attempt was made to note the analogies between these examples and the most frequent code behaviors, in order to take advantage of the already known programming patterns. For instance, this made it possible to make the first improvement by transferring some data from the global SDRAM memory to local block RAM memory, on-chip, with lower access time.

After, some attempts were made mostly to the point where a slight modification of the algorithm was necessary. In each subsection of chapter 4, there is an explanation of the changes made and partial results. Various tables will be presented depicting the estimated latency achieved and the use of resources. During the treaty, reference is made to several FPGA models, and in the final parts, a comparison will be made between them as regards the final version of the code.

When the near theoretical maximum of the obtainable performance was approached, the code will be finally tested on a real hardware platform, meeting other problems not foreseen by the previous steps. These mostly concerned the latency of access to global memory and the interface between host and device. Everything will be displayed in the final chapter, reserved for future work and some brief considerations on the results.

Chapter 2

5G: a nascent technology

There is no question that 5G has been a trending topic in recent years. The new mobile network technology is expected for the significant impact it will have. Several fields and areas will benefit from it. Not just private networks: corporate networks, public network and infrastructure, secure communications, critical communications, industry, to name a few of them.

The commercial one represents the driving force behind 5G, given the wide variety of applications in areas such as energy, agriculture, health, transport, smart cities, smart society, and smart factories [2].

The mobile network requirements to support these emerging applications are manifold. Academic and industrial studies [2] revealed among these:

- Data rate of $1 - 10$Gbps: an increase of at least ten times compared to the 150Mbps peak of the 4G-LTE.
- Round trip latency of 1ms: a ten times reduction of round trip time from the 4G-LTE one, settled around $10ms$.
- High bandwidth per unit area: increase in the bandwidth necessary to ensure a vast number of broadband devices connected within a specific area.
- Perceived availability of near 100%: the network must be available for better user experience and always in case of highly critical applications.
- Full coverage: increased and complete coverage even in the most difficult to reach areas.
- Increasing number of connected devices: to support the vision of the IoT and beyond. Devices will become a central part of the new mobile network technology.

- Longer battery life: adoption of new network protocols to reduce the power consumption of mobile devices.

- Reduction of energy used: 90% cut for a low environmental impact technology.

5G is defined by a set of specific standards agreed by entities, associations and global partners such as 3GPP, ITU-T, and ETSI. The 5G NR (New Radio) specification, i.e., the RAT (Radio Access Technology) on which 5G is based, was developed by 3GPP with multiple consecutive releases [3] (14 and 15) between 2017 and 2018, at the end of which the standalone version for NR was released.

The transition from 4G to 5G has been accelerated by exploiting the existing 4G-LTE infrastructure [4], according to what it is called non-standalone. The new services are, in fact, made available in existing 4G networks using mobile terminals capable of employing the connectivity to 4G-LTE base stations and 5G NR [4] ones. By standalone mode, on the contrary, it is meant the use of its own and renewed infrastructure, updated following the new technical specification.

The technical innovations supporting the non-functional requirements listed above are also various [5]. These are reported below.

Use of higher frequency bands and mm-waves NR supports existing mobile bands (below 6GHz) and wider bands (above 24GHz), a range of frequencies to be identified with the name of mm-waves (millimeter waves). The size of the channels for higher frequencies grows up to 400MHz. 5G capabilities are fully exploited thanks to these broader bands.

The need to use additional frequency spectra must inevitably lead the systems to guarantee the coexistence of bands with different propagation characteristics. Communication at these higher frequencies is affected by a more significant attenuation of radio channels, which often results in limiting the network coverage. The use of mm- waves, in fact, limits the propagation of the signal through outdoor environments. More dense deployment of transmitters and receivers is required in areas with high data demands - such as railway and subway stations, malls, offices - and devices, the latter acting as small relays in the new architecture. The preferred type of communication is, in general, LOS (Line-of-sight) over NLOS (Non-line-of-sight) [2].

Thickening of the existing cellular network 5G aims to add a large number of small cells and low-power nodes to support P2P (peer-to-peer) communication, like D2D (device-to-device) and M2M (machine-to-machine), significantly expanding the architecture of the previous RAT technology. The 5G network will be a multi-tier heterogeneous network, with a much higher density than the conventional single-tier network based exclusively on macrocells.

The design of cellular networks has historically been based on the role of the *cell* as a fundamental unit within the radio access network. A device can obtain services by establishing a down-link or up-link connection, carrying both data and control traffic. 5G sharply changes this picture, requiring significant modifications [6].

The innovative scheme of the 5G wireless network breaks the continuity of the previous ones based on the centrality of the BS, moving towards a topology centered on the device [2]. The use of increasingly intelligent devices will impact the mobile network: D2D and smart caching, for example, will lead to architectural redefinition becoming fully device-centric. The process happens through the acquisition of the ability to communicate by exchanging information through multiple sets of heterogeneous nodes [6].

Support for FD (Full-Duplex) communication FD enables simultaneous transmission and reception on the same frequency and at the same time. Up to earlier technology, it was commonly assumed that a node could not receive and transmit at the same time due to internal interference [5].

With advances in antenna technology, and with the support of advanced RF cancellation techniques, FD transceivers (transmitters and receivers together) are now used, which potentially double the transmission efficiency.

Use of M-MIMO (Massive Multiple Input Multiple Output) M-MIMO uses antenna arrays containing hundreds of antennas capable of serving different user terminals/equipment at the same time at different operating frequencies. This new MIMO technology depends on the superposition of the coherent phase signal starting from the numerous antennas of the base stations.

Capacity grows thanks to the exploitation of spatial multiplexing and the sum of the various contributions deriving from antennas and repeaters [7]. Adding antenna elements helps to emit more rays; this entails a higher throughput within the same area and the possibility of serving more users with the same antenna with advanced multiplexing techniques.

Use of beamforming Beamforming happens with the contribution of the M-MIMO and the antenna with adapted steering. The adaptability of the beam is essential to achieve robust mobile communications in the range of mm-waves in changing environments, such as the different common propagation scenarios taken into account by channel models, which will be seen later.

Rising energy efficiency More amplifiers are now used to supply power to the more numerous antennas, but less expensive, allowing the replacement of old equipment and gaining not only in performance but also in energy efficiency [7].

Use of C-RAN (Cloud-based Radio Access Network) At the network level, one of the most significant events is the movement of data to the cloud so that it can be accessed from anywhere and from a wide range of different platforms. This practice actively redefines the way endpoints operate [8]. C-RAN will lead to the final decoupling of the transmission and data processing equipment through virtualization [6].

Virtualization of wireless resources The reference technological innovations, wanting to deepen this aspect, are NFV (Network Function Virtualization) and SDN (Software Defined Networking).

NFV [8] allows classic network functions, traditionally delivered by hardware equipment, to run on cloud infrastructures inside data centers. However, this may mean that slightly different solutions will have to be implemented compared to existing cloud commercial solutions. A milestone for what concerns the future of the network will be this separation of network functions from the traditional hardware infrastructure, with the so-called separation of data and control plane.

SDN is an architectural framework for creating efficient programmable networks [8]. It is defined as an architecture where control and data planes are separate, the intelligence of the network and the status are centralized, and the application abstracts from the underlying network infrastructure. The main advantages lie in the programmability of the network through exposed services, which allows the manipulation of the capabilities of the object network.

Figure 2.1: A 5G suburban base station (Adapted from [9])

An accurate test of a rising technology like 5G is as mandatory as it is significant

in terms of efforts and costs. The test phase of a mobile network must necessarily cover the complete cycle of a technology, from the primary engineering phases to the laboratory ones, continuing for those of support to the operators, finally reaching the benchmarking and monitoring of the quality of the service offered [3]. Often, the challenge is to do it before the launch of a standard, with a productive simulation phase.

The simulation and implementation of propagation models, as a preventive section of the testing phase, becomes increasingly important in wireless communication systems. Tools such as channel simulators are hence essential for performance estimates, evaluations, and decisions in a network deployment perspective and planning, before definitively embracing a new technology [10].

Radio network planning is a fundamental step of any nascent communication technology. All industries and mobile service companies do planning at their discretion. Whenever a new cellular technology is chosen for future mass use, thousands of RF parameters are adjusted in order to provide optimal and appreciable values. It is easy to realize how this phase can be costly and how it can take considerable time, more than the one available, dictated by precise market windows.

Before the commercial use of an outgoing technology, this tuning phase can be significantly facilitated by simulations. These bring meaningful benefits in terms of costs, above all, replacing heavy physical equipment with accurate software models [11].

Today network planning consists of several phases: an initial phase of collecting pre-planning information on the territory, a phase of pre-sizing the network at an infrastructural level, link budget preparation, simulation of the calculation capacity, and coverage. Only after these stages, it is possible to move on to the selection of physical devices and their positioning for expected coverage and performance [11].

2.1 Benefits of channel model simulation

As mentioned, the measurement and modeling of a propagation channel represent the first step in the use of new mobile technology and the preventive steps for testing the RF modules - transmitters and receivers -, BSs (Base Station) and UEs (User Equipment). The critical challenges for channel measurement and modeling concern [5]:

- Efficient and realistic measurement: measurements must support the extension of current propagation models by capturing different frequency ranges. The output data must be consistent.
- Spatial distribution and mobility: new paradigms such as D2D/M2M have to be taken into account; also, realistic cases such as indoor/outdoor, crowded

areas, vehicle-to-vehicle communication, and others should be considered. It is crucial that this contingent shift of mobile terminals and network nodes is modeled for each link.

- Arrays of large-scale antennas and mm-waves: used for directional communication, the channel model must consider these new implied technologies, improving angular resolution, supported spectrum, gain measurement after signal multiplexing.

By channel model is meant the model of the medium interposed between typical transmission and receiving stations, both equipped with one or more antennas [12]. A radio channel consists of two antennas, one transmitting and one receiving, each alongside a propagation channel. The propagation channel is the environment in which radio waves propagate from the transmitting antenna to the receiving antenna [13].

What a model aims to do is to reflect physical reality with a level of detail appropriate to the purpose, balancing accuracy and complexity [13]. The design of a channel model starts with the requirements of the target system to describe. Also, and the purpose to which the final model will have to fulfill is an important parameter.

Whereas in the past channel models depicted only 2-D spaces, the majority of simulation models now depict fully 3-D spaces, for both transmitting and receiving antennas, being able to evaluate more complex antenna structures and propagation scenarios [14].

The characteristic of each radio signal varies in passing from the transmitting point to the receiving point in space. In particular, this may depend on the distance between the two stations, on the environment such as trees and buildings, on other variables such as carrier frequency, type of encoding, bandwidth, Doppler frequency, polarization, weather conditions, and noise. Fading is a central feature of a radio channel model, as it considers the fluctuation of gain as a function of time, frequency and space. [13].

These are the most typical effects that influence propagation. The simulation models must cover all the various techniques and propagation environments. These are often complex because they are used to model very different scenarios. Models try to reproduce the chosen scenario realistically, often facing trade-offs in terms of accuracy, generality, and simplicity. [14].

The main drivers for the construction of 5G channel models are multiple antennas, new frequency bands, support for mm-waves, new transmission and reception scenarios of devices. The construction of the 3GPP channel model briefly presented later supports the reflects a consistent multi-dimensional radio channel from 0.5 to 100GHz. By consistent, it is meant that it takes into consideration the signal conditions for band differences, cross-band interference and that it is able to

correctly correlate the output data to the model [13].

A mathematical representation of the effects these various parameters bring to the propagated wireless signal is the channel model. Technically, this can result in the calculation of the impulse response of the channel medium in the time domain or the Fourier transform in the frequency domain. Modeling a radio channel brings tangible advantages in network decisions and planning, including connections optimization and choice of compromises in terms of performance to be used.

2.2 Link layer simulator design

The thesis work concerns the optimization of the NR channel model used in the link-layer simulation. The model used in the design and reported in this chapter is fully described and calibrated in [15]. It is a generic model supporting frequencies between 0.5GHz and 100GHz, with bandwidth up to 10% of the central frequency and, in any case, no more than 2GHz. This model is valid both for system simulation and for link-layer simulations.

As far as the system-level simulation is concerned, this is substantially new to the past dictated by the emergent network architecture and the essential technologies introduced. First, it must be taken into account that much more space will be needful for internal data storage, deriving from much more complex parameters, more detailed scenarios, and new evaluation metrics [16]. Second, the calculation of interference becomes considerably more sophisticated in the new heterogeneous system.

Consequently, the construction of a system-level simulator must evolve in various directions, to solve well-known problems. Some of them are [16]:

- Propagation channel modeling: with arrays of numerous antennas to equip the base stations, the channel's characteristics must be modeled in totally different ways.

- Efficient processing ability: vast volumes of data must be processed within the simulation, which makes it necessary to have considerable computing and storage capacity.

- Complex interference simulation: interference increases significantly by introducing M-MIMO and FD, and these must be described carefully.

- Interference Calculation Model: the heterogeneous and ultra-dense network structure differs widely from previous models in the method of calculating simulation parameters.

- Additional clustering: clustering methods are considered in the case of networks composed of small cells.

For what concerns the simulations at the link level, instead, which is the one of interest for the thesis work, the model is of the typology defined as SCM (Space Channel Model) and simulates the UE-gNB connection by generating the representation of the link in between.

A UE is any device adopted by the user when receiving; this can be a smartphone, tablet, or laptop, equipped with a mobile network adapter. By gNB, it is intended a 5G base station that provides NR services for user and control planes.

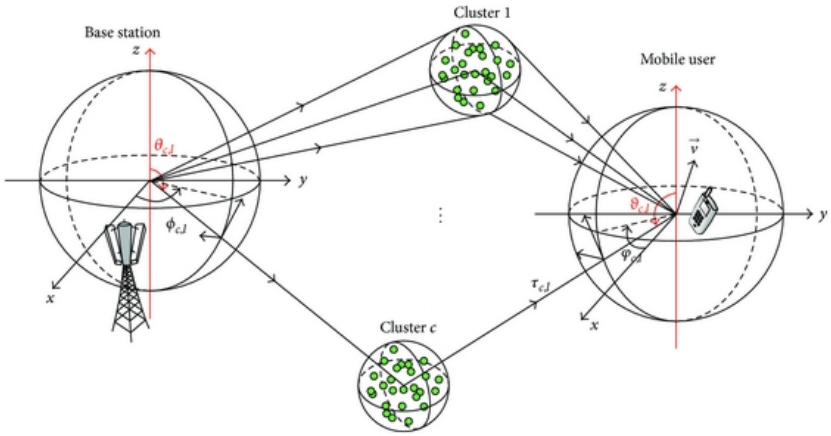


Figure 2.2: Generic 3D fading channel model (Source: [17])

Two types of channel representation are possible: one relies on the TDL (Tapped Delay Line), the other one on the CDL (Cluster Delay Line). The TDL description, for simplified evaluations, is based on the impulse response of the channel. With the TDL model, a radio channel is specified by several delay taps, each with its characteristics. These can be the average power, the delay of the propagation path, the distribution of the amplitude of the signal [13]. The resulting radio channel will be a fading model in the time and frequency domain, based on statistics. The CDL description, instead, is based on the departure and arrival directions of the signal in space: this allows a better representation of beamforming since the signal direction in a three-dimensional space is taken into account.

The advantage of the TDL model lies in the correlation between antennas, defined by a static correlation matrix. The CDL representation, on the contrary, strongly depends on array geometry and in the way the channel propagates.

The creation of the radio model takes place by following the steps reported in [15, subsection 7.7.1]. In that paragraph, it is accurately reported how three

different models are built respectively for the NLOS case, CDL-A, CDL-B and CDL-C, and two other models are built for the LOS case, CDL-D and CDL-E.

The CDL model, as mentioned above, is based on departure and arrival spatial directions; these are generated in the first step of the procedure.

After two further steps relating to the coupling of the rays and their polarization, it is the time for the generation of the coefficients; for this, please refer to [15, section 7.5], defined as Fast Fading Model.

The impulse response of the channel is computed considering all the previous inputs. Once this response is known, it is then applied to the channel.

The generation of the coefficients and the application of the impulse response represent the computationally burdensome operations, which need to be hardware-accelerated. This topic will be discussed later in the thesis with explicit references and precise estimates of performance.

Chapter 3

Hardware acceleration based on FPGA

Field Programmable Gate Arrays play a crucial role in the digital circuit market. These are prefabricated silicon devices that can be reprogrammed according to the most diverse needs, practically becoming any desirable digital system or digital circuit. The increasing flexibility and access to multiple functionalities of these devices open the doors to the execution of various activities and to be modified directly on the field [18].

For medium and small businesses, an FPGA provides more valuable and cheaper access to digital design. An alternative resides in the so-called ASICs (Application Specific Integrated Circuits), i.e. custom circuits explicitly created for a specific purpose and not reprogrammable. Typically, the manufacturing time of these devices is months, and the costs for the masks are incredibly high: the use of these circuits is also unthinkable if not massive and vast markets where it is possible to share the costs with the total number of the entities to sell.

On the contrary, an FPGA is quickly programmable, and the evolving capacity of the integrated circuits meets the designers allowing today extensive programming that approaches that of an ASIC. The technological evolution regarding transistors has reached a limit. The price no longer drops as predicted by Moore, the computing capacity of processors will no longer grow following the trend. FPGAs become of great help to traditional CPUs, with their excellent capacity for task parallelization.

GPUs (Graphics Processing Unit), born to assist processors in modern computer architectures, can also prove to be the chosen one for the hardware-acceleration covered in this work. These, as will be reported later, are intended to carry out tasks with the least possible logic, where all the operations to carry out are in parallel (i.e. operations on multidimensional arrays). FPGAs, from this point of

view, also leave room for a part of application logic that is not suitable for GPUs. However, GPUs are very suited for fast floating point operations, while FPGAs are more committed to fixed point ones.

A fundamental aspect that makes the FPGA a competitor is the low energy consumption.

From the architectural point of view, an FPGA consists of a high number of functional blocks of different types. These types include generic logic, memory, multiplier blocks, all supported by a programmable routing fabric for the interconnections. Finally, input/output blocks allow connecting the chip with the outside.

The most common technology that provides FPGA programmability is based on SRAM [19]. A static memory cell contains up to 6 transistors and conforms to CMOS technology. An SRAM cell can be programmed an indefinite number of times, and this is another critical point. For previous flash-based technologies, reprogramming was limited due to the non-durability of the components linked to electrical phenomena. In the case of fuse (or anti-fuse) technologies, programming was reduced to one time only.

The disadvantage lies in the volatility and the size of the memory: SRAM needs an external device that performs its programming after being switched off; furthermore, being a single cell made up of 6 transistors, the size grows. Another possible combination is to use a flash memory together with SRAM programming technology. Generally, devices use on-chip flash memory to provide non-volatile storage and SRAM cells to control programmable elements in the design.

As for the logic called *soft-logic*, precisely the programmable units, this focuses on LUTs (Lookup Tables). A LUT is a tiny memory capable of generating essential Boolean functions, such as OR, AND, XOR, NOT and some combination of them. By adding other components such as a FF and a multiplexer to a LUT, a CLB (Configurable Logic Block) is obtained.

CLBs represent the primary resource in the implementation of general-purpose combinational and sequential circuits. They execute the logic, being programmed to reconfigure the logic gates, as well as they implement memory functions.

Figure 3.1 shows the basic architecture scheme of an FPGA. White blocks represent the programmable I/O, while grey blocks the CLBs, the actual configurable logic. All the interconnections between these, the paths in the figure, are also programmable. This is called routing.

The modern FPGA modules include also other more specific blocks in order to improve even more the computation and parallelization capacity. As far as the *hard-logic* is concerned, that is constituted of components such as DSP (Digital Signal Processor), BRAM (Block RAM) and various memory controllers.

The DSP is the most sophisticated computational block, specially designed to accelerate typical problems related to digital signal processing. It is, in short, an ALU (Arithmetic Logic Unit) composed of some registers, one or more adders, a

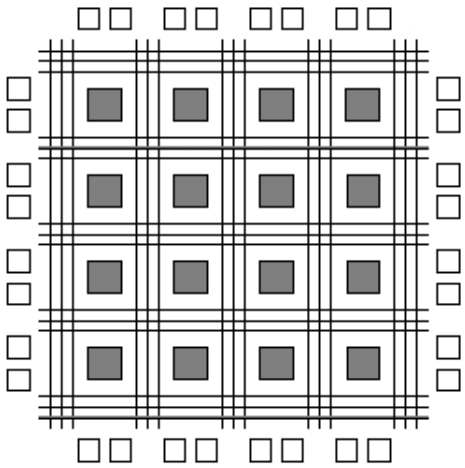


Figure 3.1: Basic structure of an FPGA (Adapted from [20])

multiplier and a pattern/sequence detector in order to speed up repetitive sequences and bring significant improvements to complex operations such as accumulations. DSPs are often used for the implementation of logical functions that would otherwise presuppose an overly extensive use of the LUTs.

By BRAM it is meant the high-speed local memory present on the FPGA. Each memory block of this type is true dual-port and can be of different sizes depending on the model. Multiple blocks can be connected to meet space requirements, in case, for example, a buffer overcomes the size of a single memory block. The leverage of this memory will prove to be crucial in accelerating the problem discussed in the thesis.

The higher speed increase that can result from the use of an FPGA does not lie in the mere clock rate (frequency) but in the possibility of parallelizing the functional units [21] that make up the programmable device. The fundamental parameters related to the definition of speed are three [22]. Which one to refer to depends on the problem at issue.

The first is *throughput*, i.e. the amount of data processed at each clock cycle, measurable, for example, in bits/s. High throughput architectures maximize the number of bits per second processed by the application.

The second parameter is *latency*: this is the time delta in terms of cycles between an input data and the same output data once processed. Low latency architectures allow minimizing the delay between the entry of the system and the exit.

The third and final parameter is *timing*: this refers to the logic delays between sequential elements of the circuit. Timing is measured in clock period or frequency (1/period).

The optimal result is to maximize throughput by minimizing latency and respecting the preset clock period. It is necessary to exploit the execution of multiple instructions in a highly parallelized manner in order to achieve this performance. This concept is called *pipelining*.

Pipelining is based on the simple concept of accumulating instructions or tasks which will then be carried out in an orderly manner. Each step of the pipeline does a different job. Many books exemplify the concept by taking it easy in a simple laundry room. More in general, any production line can be thought of as a sequential pipeline execution.

In a domestic laundry, there is a basket of dirty clothes, a washing machine and a dryer. It is impossible to proceed in parallel on the same clothes (input data): these cannot be washed and dried at the same time. However, it is possible to make use of all three stations considered with three different baskets of clothes (different input data). The basket is filled with dark coloured clothes, then dipped in the washing machine. Meanwhile, the basket is filled again with light coloured clothes. When the latter goes into the washing machine, the dark clothes will be drying, and the basket can now be filled with other clothes. In this way, each of the three steps of the pipeline is used simultaneously maximizing throughput.

The goodness of a design that exploits this mechanism is that new data can be processed before the previous data has finished. Pipelining is used in all high-performance devices, including CPUs. The stages in question are the way processors commonly handle instructions, i.e. fetching, decoding, executing (and accessing memory). Leveraging the pipeline is mandatory in order to achieve the desired performance also on an FPGA. The stages, in this case, are defined on the source code and the instance of resources necessary to carry out the required operations.

3.1 High-level synthesis

Until the late 1960s, integrated circuits were designed, optimized and configured by hand [23]. With the advancement of new technologies related to silicon and the ever-increasing complexity of applications, differences in the design methods and supporting tools have been embraced.

One of these changes concerned the transition from the RTL level description to the so-called HLS (High-Level Synthesis). Through a higher level of abstraction, it allows the generation of RTL logic optimized in terms of performance, area and dissipated power, as well as a visible reduction in design times compared to HDLs (Hardware Description Languages) [24]. RTL is an abstraction whereby synchronous logic circuits are described by means of a state machine, and the passage of digital signals takes place at the level of hardware registers. Hardware description languages such as VHDL or Verilog describe digital circuits at this

level of abstraction. As circuits grow in complexity, this technique becomes indeed lengthy and error-prone.

The advantage, therefore, does not lie in the shortest time for the design itself, but at the time gained and exploited in the optimization guided by advanced and advanced tools exploring the design space.

A further advantage consists in expanding availability also to non-specifically hardware designers, making design possible even for software engineers through the supply of complete tools [25]. Despite potential benefits such as energy and performance, hardware design is often too complicated, expensive and limiting, which is why a software approach is preferable for most applications. Hardware description is often very heavy and bulky [26], with a consequent tendency to not fully comply with the initial specification.

First, an HLS tool compiles the specification. The input languages notoriously supported are C, C++ and SystemC, but this may vary depending on the tool. MATLAB is sometimes used also .

The compilation phase includes the classic steps: a syntax check and other changes desired to optimize the code given to it. An example may be the elimination of code never performed (in the event of conditions never verified), the replacement of macros, or the elimination of false dependencies.

Secondly, it allocates the necessary hardware resources such as logic and memory blocks, functional units and buses.

The scheduling phase follows, i.e. the assignment of operations to the clock cycles. These operations are subsequently linked (so-called binding phase) to the functional units capable of executing them, just as the variables assigned to specific memory locations.

Finally, the RTL architecture is generated [23].

Some well-known HLS tools on the market are Mentor Catapult, Cadence Stratus, Altera Quartus and Xilinx Vivado. As will be covered in detail later, the tool used in the following work is Xilinx Vivado HLS.

It is an advanced software tool capable of directly programming Xilinx devices without having to manually resort to RTL design, operating on high-level specifications (C and C++). In combination with the OpenCL framework, which will be discussed in the next section, this allows the automatic use of the resources of the target FPGA, with an enormous gain in comfort. It is able to use on-chip memory and DSP elements automatically and to conveniently map floating-point operations through specific libraries.

It also allows to abstract from many other implementation details.

3.2 OpenCL: an overview

Xilinx tools allow the use of C / C ++ and OpenCL languages. OpenCL is a parallel programming framework by Khronos Group, recognized as an industrial, portable standard, used for programming heterogeneous and multi-core [27].

For heterogeneous platforms, we mean a CPU together with other hardware components/accelerators such as FPGAs or GPUs, as in our specific case. The trend expressed well by the so-called *manycore* has been, for a decade or more now, that of pushing towards the execution of extra computational elements in parallel instead of a single or few powerful elements. This tendency led to the research (and invention) of programming paradigms that would allow the mapping of complex algorithms on parallel architectures without massive efforts by programmers [28].

OpenCL provides a programming language and an API to support programmers, offering numerous low-level details about the hardware in use and various ways to exploit them. The one just described is the real convenience of OpenCL, that is, the possibility of exposing the problem and the need for parallelism to an underlying automatic tool.

The synthesis tool, perfectly integrated with the environment, adapts OpenCL to the design flow by efficiently mapping an application in an FPGA allowing its execution in parallel.

Heterogeneous systems are increasingly adopted for their superior performance compared to homogeneous systems, at non-prohibitive costs. Over time, accelerators continue to grow and become more powerful.

OpenCL represents today the most accepted standard in this field. Although OpenCL focuses on data parallelism, activities can also be carried out in parallel in the framework, with execution distributed across multiple devices. The user, in general, can create a queue of tasks (called *running queue*) and assign these to different computing units; this means that a user can launch multiple activities, already paralleled internally, in parallel on multiple units.

The following are the models of OpenCL regarding platform, memory and execution.

Platform model An OpenCL platform is the mixing of a processor referred to as host and one or more computing hardware devices.

The host processor is responsible for the execution of the host-side code, together with the normal startup and launch activities of the OpenCL framework and driver management. Additionally, it manages the task of configuring data transfers between host and device using global memory buffers.

The device is instead the hardware element on which a computing kernel is run. With kernel, it is indicated the code that will be executed on the device.

An OpenCL kernel is essentially the part of the application parallelized to its maximum [28] aimed at obtaining the best performance. Each device can be further distinguished in separate compute units, and again each unit divided into processing elements. A processing element represents the fundamental block for the execution of the work elements.

In our case, the device is the FPGA. The latter is initially configured as an empty calculation unit. It is up to the user deciding the level of customization to support a single application or a class of them.

Memory model OpenCL provides a hierarchical representation of memory. This configuration is present in all OpenCL implementations and versions, but the mapping of model and hardware is up to the different suppliers. As regards the Xilinx SDAccel platform, i.e. the one that will be used for the FPGA optimization and programming work, this is divided into the following, as specified in [29]:

- Host memory: this is the system memory only accessible by the host processor. The data needed by the kernel must be transferred to and from global memory through a particular OpenCL API.
- Global memory: this is a memory accessible to both the host processor and the device, although this occurs at different times. The host initially deals with the allocation of the necessary memory buffers. Host applications access the read and write memory until the kernel execution is started; from this moment and until the end of the kernel execution, the device is the only one to have access to the global memory (always in read/write). Once finished, the host will again be able to access memory.
  There is a subsection of the global memory defined as constant: this is accessible in the manner indicated above as regards the host, while access by the device is limited to read-only functionality.
- Local memory: this is a memory section that cannot be accessed by the host processor but only by the device. It is typically used for storage of data shared by multiple work items.
- Private memory: this is the area reserved for a single work item. As with the previous one, the host does not have access to it.

Execution model The OpenCL execution model defines what the mode of execution and launch of the computation kernels is. It is based on the concept of NDRange (N-Dimensional Range) or index space. Imagine this to be a simple line (1D), a square (2D) or a cube (3D). A number of points compose this line,

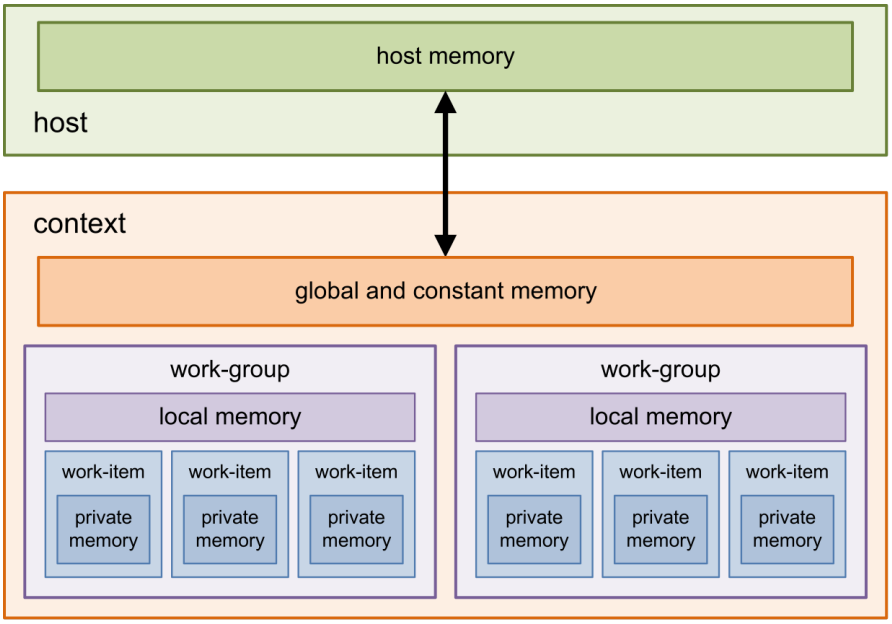


Figure 3.2: Memory model exposed by OpenCL (Source: [30])

square, or cube; each of these points is filled with a different kernel with a unique ID. A single kernel instance in the index space is called a work-item: a work-item executes the same compute kernel on different data.

It is important to note that a device is free to perform various work-items in parallel. When a user sends a kernel to a device, the NDRange has to be provided. This topic will not be explored in the thesis work, focusing in particular on optimizing a single kernel and executing it as a single work-item, nor in parallel with other work-items. Hence, the provided NDRange parameter to the tool will always be one dimensional, and a kernel instance never executed in parallel with other instances. The work consisted of maximizing the performance of a single kernel, trying to force the operations to be parallelized and pipelined where possible. Therefore, the coexistence of multiple kernels was not taken into account, nor different kernels or the same kernel on different data in parallel. That would have required a severe change in the code, and more generally, in the whole simulation. It has been agreed with TIM to proceed with the optimization of the code, not considering a significant reinvention/rewriting of it.

Chapter 4

Thesis work

This chapter focuses on the actual work of the thesis. At first, the point from which the development of the work has started will be exposed. That is the results of a previous master's thesis.

All the significant steps in obtaining the results will be exposed, with specific evaluations about the aspects deemed essential or fundamental. Particular attention, as will be seen, will be put on procedures aimed at improving the existing code to be hardware accelerated.

An in-depth comment on the results obtained, comparing them with the desired ones, can be instead found in the next and final chapter.

4.1 Starting point

The initial code for the channel simulation provided by TIM is a hybrid code consisting of both MATLAB and C++ code. The coexistence of the two languages is possible with an API called MEX. MEX is a MATLAB function witch can call a C/C++ routine. The two are often integrated into a wide variety of applications, e.g., in signal processing. This integration is advantageous since C++ is compiled, and therefore usually faster than an interpreted language like MATLAB.

Originally the code was entirely in MATLAB, and the results were disappointing in terms of performance. That was mainly due to the difficulties in exploiting the processor on the machine where the simulation ran.

As reported by the TIM work supervisors, the transposition of the computational part into C++ has led to a noticeable drop in simulation time. That improvement allowed the exploitation of the floating-point unit of the Intel CPU on their servers, of which there are no further details.

The MATLAB code, at present, does nothing but the initialization of the parameters. Despite this, the simulation time remains long, hence the request for

appraisals on the possibility of accelerating using proper hardware.

In particular, the choice fell on FPGAs. TIM thought to evaluate the effects that are obtained from hardware acceleration, starting, in fact, from the use of an FPGA. The same evaluation could be done using GPUs in the future, possibly comparing the results. The code reports a mixture of pure and simple operations on double-precision floating-point data and logic. By definition, FPGAs represent a good compromise in similar cases where execution can vary and depend on logical statements. On the contrary, GPUs are mostly indicated in the case of simple operations on floating-point data - like arrays and matrices -, highly parallelizable, where logic is almost absent.

Figure 4.1 reports all the channel model's simulation steps.

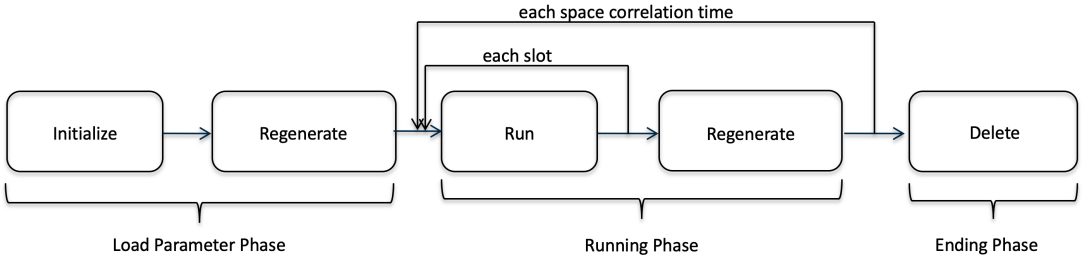


Figure 4.1: Channel model simulation steps

The initialization phase, as already mentioned, takes place in a MATLAB routine. This function concerns the initialization of the simulation parameters, which include: type of representation of the cluster among the models briefly discussed in section 2.1, type of delay, number of antennas that transmit and receive, sampling frequency, and other specific parameters. These are used to differentiate how the samples are processed during the simulation.

The regeneration phase is called into action for the recalculation of channel parameters such as correlation, angles of arrival/departure of the signal, and the position occupied in the space of the user terminal. In this phase, the computational heavy part of the channel simulation takes place. First, the coefficients relating to the impulse response of the channel are computed. This is the function accelerated by N. A. Shah in his thesis [1]. Further details will be reported in section 4.2. Then the function is applied to the inputs, and consequently the outputs are collected; please mind that this it is the central function that will be accelerated in the thesis work. As for the arrows in the picture, a summary description is now given. The simulation is composed of slots, based on a variable parameter. At this time and in the continuation of the thesis, this parameter is set to 10. This means that the function is called to perform its task ten times. Same code, different data. When the function execution time will be reported in the next sections, this will be the

average of the ten execution times. As for the more external arrow, TIM stated that for this NR link simulation, the space correlation time is maintained for all the simulation. Hence it will not add additional cycles to the current simulation.

The final step is to free up the memory space at the end of all the performed steps.

The next section briefly presents the work done by Nasir Ali Shah.

4.2 Previous work

The previous thesis work consisted of two parts.

The first part involved the removal of the MATLAB code in favor of the final C++ code. Therefore, a separate version of the C++ code has been created, including the whole initialization phase of the data structures and the call to the functions previously carried out by MATLAB.

The reason why it was chosen to isolate the environment is twofold. First of all, this leads to the security of having an optimized code. With the MEX file no longer brought into question, everything happens in the new C++ code, and the data structures are filled efficiently. Secondly, this somehow overcomes the interfacing constraint, a potential bearer of many problems when this two-actor system (MATLAB and C) is integrated with the new heterogeneous software/hardware system of the OpenCL environment. In other words, a difficulty of interfacing between the MATLAB initialization phase and the subsequent call of the hybrid execution code (host and device) was found.

It was preferred to tackle the problem later, once the actual possibility of obtaining an advantage with the help of hardware was verified and ascertained, being the specific purpose of the work. The compatibility problem will be addressed as one of the points in the future work, in the final code packaging phase.

The second part had to do with the transfer to hardware with subsequent optimization of a first function identified as critical, in terms of execution time. This is used to calculate the TDL (Tapped Delay Line) coefficients for identifying the response to the channel impulse. That part is essential being what precedes the accelerated function in this thesis. The discussion will not go into further details, which are reported in [1]. The procedure for accelerating the part was similar to the one adopted for this thesis.

During the work of this degree thesis, the pure C++ code rewritten and readapted has been exploited, thanks to the help of Nasir. The union part of the two accelerated functions, so the two OpenCL kernels, will be carried out in the future work.

4.3 Overview of the problem and simplified code

As already mentioned in the previous sections, the work focuses heavily on the assessments linked to a precise section of the C++ code provided by TIM. That one is the part of the code responsible for applying the impulse response over time to the input symbols by obtaining the output symbols.

Simplified code fragments similar to the original will be issued in this chapter and later on. The crucial part consists of four nested loops, to which it was convenient to affix labels. This practice is common and beneficial because, in the various Vivado HLS reports testifying the progress, each cycle will have its name, making it easier to be recognized. That will, therefore, be appreciated later when the results between various code versions will be compared.

The simplified code structure is shown below. It can be seen as a top view of the target code to accelerate. No operations or any logic within the cycles are reported here.

```
rxAntennaLoop:
for (int rxIdx = 0; rxIdx < 2; ++rxIdx) {

    txAntennaLoop:
    for (int txIdx = 0; txIdx < 32; ++txIdx) {

        coefLoop:
        for (int cdlIdx = 0; cdlIdx < 23; ++cdlIdx) {
            // compute coefficients for interpolation
        }

        symbolLoop:
        for (int s = 0; s < 122880; ++s) {

            interLoop:
            for (int i = 0; i < 23; ++i) {
                // do interpolation
            }
        }
    }
}
```

Note how high the total number of iterations is. It consists of $2 \times 32 \times (23 + 122880 \times 23)$ cycles. In fact, an output symbol - output of the innermost cycle defined as "interpolation" - is produced or updated:

1. For each receiving antenna, for a total of two times. Two are in fact the antennas receiving the signal in the simulation based on what can be seen

from the initial configuration. The number of receiving antennas, as well as those in transmission, can be changed.

2. For each transmitting antenna, a total of 32 times. The antennas which transmit are 32 in this simulation.

3. For each connection at each sample time, for a total of 122880 times.

4. For each cluster, a total of 23 times in the case under consideration. This number depends on which type of channel you want to simulate. In this case it is the CDL-B [15, table 7.7.1-2] which includes 23 clusters.

As can be seen below, some adjustments have been made, but there have been no real distortions of the algorithm. Such manipulation would have meant going against the nature of this work.

The thesis aims to obtain the maximum acceleration possible with the knowledge possessed at the beginning and acquired in the course of the work, with limits in terms of time and resources provided. The work wants to estimate the possibility of an improvement through multiple models of accelerators on the market with considerations on some involved aspects and critical issues encountered.

The computational complexity of the function derives from the nested cycles that give shape to a very high number of overall cycles. Within these, ordinary arithmetic operations and accumulations are performed on double precision floating point data.

Another aspect that will be critical is related to the arrays of data feeding the simulation. The transfer of these data from global memory to the BRAM memory will be fundamental for a first significant improvement, as will be seen later.

As long as these are very large datasets, it is not plausible to attempt to speed up the execution with legacy FPGA models. This is due to the fact of not being large enough to keep useful data into on-chip RAM for more efficient use. The positive aspect of having big datasets is to have concrete estimates of the use of hardware such as FPGAs in realistic work cases.

The high number of cycles and the complicated access to memory limit the synthesizer in finding a way to speed up the execution, performing its operations in parallel.

4.4 Software performance

In this section, the performance of the pure software version of the code executed on CPU will be evaluated. The results will be taken as a reference to evaluate the speedup obtained by the hardware in the next sections, and for the final results of the last section.

Two times from two different CPUs have been taken as reference. First, the simulation was performed on a server at Politecnico. Below are reported the model and execution time relating to the specific function of the full simulation code to be accelerated.

- CPU: Intel® Core™ i7-6900K @ 3.20GHz;
- Average execution time: 5.01s.

The second machine on which the code was compiled and executed is the Amazon EC2 server, where the final work will be tested. As in the previous case, the CPU model and execution time is shown below.

- CPU: Intel® Xeon® E5-2686 @ 2.30Hz;
- Average execution time: 14.52s.

As evident, there is a remarkable difference between the two. The second time almost triples the first. Please note that the average time is computed on multiple calls of the C++ method. A simulation run contains, in fact, many calls of that function, as detailed in Figure 4.1.

For completeness, it should be added that the time on the first CPU is the lowest average time recorded in many runs of the simulation on different days, when therefore probably the load to which the CPU was subjected was low or zero. In some other runs, the average time went up to 8 - 10 seconds.

In general, such a high execution time (both the cases) is justified by the overall iterations number, along with the operations carried out within each cycle. These include floating-point additions, subtractions, multiplications, divisions, and accumulations, all operating on double-precision data types (64-bit data).

The times of access to the data are negligible, as no value changes are made with the attached copy. It is a C++ method acting on local variables inside a class, and at most on two large external arrays of symbols passed as parameters via a pointer.

There is a theoretical chance of improving the overall latency. A possible evolution could derive from concurrent programming, that is, from the exploitation of several parallel threads of execution. That should, in any case, be supported by further assessments of how to access the arrays and the probable and complex tracing of the addresses that should be carried out. This solution, therefore, falls outside the scope of the thesis.

In the following paragraphs, the same code will be adapted to the hardware architecture in the OpenCL environment, and its theoretical performance will be evaluated at first, followed by some practical feedback.

4.5 Comparison between Xilinx FPGA platforms

With platform it is meant an acceleration development board that integrates an FPGA model into an electronic board together with other components. It provides a way to connect the FPGA to the host device (desktop or server) via a PCIe interface.

Figure 4.2: View of a PCIe development board based on a Xilinx Virtex Ultrascale+ FPGA (Adapted from [31])

During the thesis, it was decided to rely on the results shown on only one of the three listed below, the first, as reference hardware for testing the code. The first indicated platform is, in fact, the one available on the AWS cloud service where the optimized code will be finally executed.

The other two platforms are respectively a slightly earlier model, with a reduced capacity in terms of overall resources, and an older generation model in which there are resource and clock frequency limits, making it not suitable for applications like this one.

The latter is reported because TIM physically owns it, so a first summary analysis concerned the feasibility using that particular hardware. Once it was learnt the real size of the kernel, however, this possibility was discarded.

Virtex® UltraScale+™ VU9P

Based on the xcvu9p FPGA part, it is the most performing and newest generation module. This acceleration development board features four distinct DDR4 channels (for a total of 64GB) and the Xilinx DMA system for PCI Express with PCIe Gen3 x16 connectivity. The number of available resources is shown in Table 4.1.

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Available	4320	6840	2364480	1182240	960

Table 4.1: Available resources of Xilinx Virtex UltraScale+ AWS VU9P F1 platform

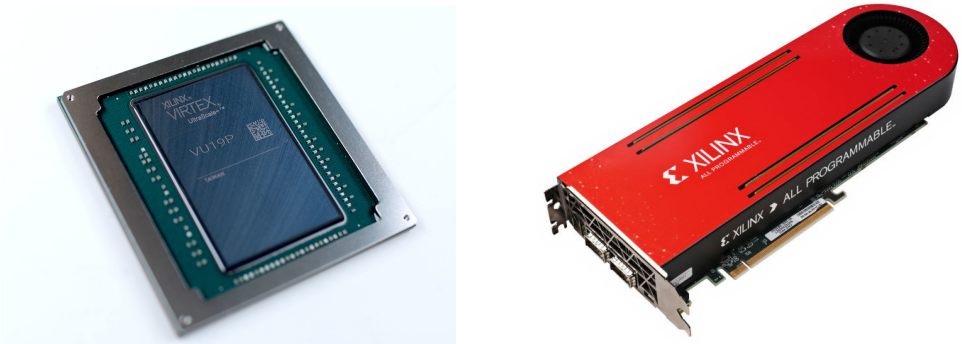
- BRAM_18K: this is the internal RAM memory of the FPGA. A block of RAM is capable of storing 18Kb of data; being the total number of blocks 4320, the overall storage capacity is 75Mb, as specified by [32].
- DSP48E: this is the number of logical elements for digital signal processing; a few words on this block can be found in chapter 3.
- FF: number of Flip-Flops contained in the module; to learn more see chapter 3.
- LUT: number of Look-up-tables; to learn more see chapter 3.
- URAM: this is a synchronous memory only present in UltraScale+™ models. It is a individually clocked dual-port memory. Ultra RAM adds 960 blocks of 288Kb each to the storage capacity, for a total of 270Mb, as mentioned in [32].

Figure 4.3 shows the packaging of an FPGA model belonging to the same series of the model in question (VU13P).

Kintex® UltraScale™ KCU115

It is a prior platform model in terms of technology compared to the previous one. The board is based on the xcku115 FPGA part, featuring up to four channels of DDR4 SDRAM (for a total of 16GB) and the Xilinx DMA system with PCIe Gen3 x8 connectivity.

Kintex® UltraScale™ FPGAs are based on a technology called SSI (Stacked Silicon Interconnect). An SSI device is capable of holding multiple SLRs, i.e., Super Logic Regions. Each SLR is formed of the common circuitry of each Xilinx FPGA, including LUTs, registers, I/O components, transceivers, multiple BRAM blocks, multiple DSP blocks, and more. These blocks will be occupied depending on the design. A focus on SLR can be found in [35].



(a) Xilinx Virtex Ultrascale+ VU19P (Source: [33])

(b) Xilinx Acceleration Development Board (Source: [34])

Figure 4.3: Xilinx FPGA module and Xilinx PCIe Acceleration Board

	BRAM_18K	**DSP48E**	**FF**	**LUT**
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680

Table 4.2: Available resources of Xilinx Kintex UltraScale KCU115 platform

The SLR, present in the Kintex® model, represents a limitation compared to the Virtex® model previously exposed. As will be seen, it can be that the percentage of total resources used is below 100%, but the percentage of use of SLR is higher and therefore exceeds the physical limitations of the device. This happens because the synthesis tool automatically reserves some SLRs for specific purposes (e.g., memory), avoiding that the same block is used for others (e.g., DSP) despite being available in principle. In any case, the percentage of use of SLRs must be monitored.

Zynq® SoC ZC706

This board was conceived for the fast prototyping of embedded applications. It is based on the xc7z045 FPGA part and features only 1GB of DDR3.

As reported by Table 4.3, it is not comparable to the previous models in terms of capacity. The throughput for the host-device data transfer is also lower, being based on past generation technology. Another limitation is the low clock frequency compared to the other two models, which is about 2.54× lower. Further considerations on this will be made later.

	BRAM_18K	DSP48E	FF	LUT
Available	1090	900	437200	218600

Table 4.3: Available resources of Xilinx Zynq SoC ZC706 platform

4.6 Code optimization

All the notable versions of the kernel, with relative latency characteristics and occupied resources, are presented case by case. Remember that, with *kernel*, it is intended the code that will be executed on the FPGA, i.e., the part of the program that will be hardware accelerated.

The method followed in seeking an overall improvement in performance varied at some point.

At the beginning of the work, a more strict procedure strongly based on attempts was followed. This made it also possible to make familiar with the work environment and the tools used. It also helped in comparing the various examples used in the initial training phase with the real case of interest.

Over time, the study of cases similar to the problem was introduced. These were mostly examples found in the Vivado SDx tool, with proven solutions related to the most common critical issues. From here, gradually reading manuals and being helped by Professor Lavagno and some colleagues, the first partial results were obtained, the goodness of which was periodically discussed with the work supervisors.

The difficulties, starting from scratch, have been multiple. In the presentation of the next paragraphs these will be stated, together with the remedies put into the field to overcome them.

4.6.1 Original code

The first step was to adapt the C++ code to OpenCL. The version supported by Vivado SDAccel and the SDx development environment is OpenCL 1.0, therefore an integrated version on C99 with some extensions. That mainly imposed only one limitation, namely having to remedy the absence of the complex type (C++ standard library) for the representation of complex numbers. These are indeed frequent in simulation.

First, to overcome this lack, a simple C structure has been created, along with the four functions of addition, subtraction, multiplication, and division, so that these basic operations could be supported. Then, on the contrary, it was decided to divide each array of complex numbers into two distinct arrays, both of data type *double* containing each and respectively the real part and the imaginary part only. In this way, a few lines were added to the code, but everything was more readable

and more controllable at that point.

Another variation immediately implemented was to transform multidimensional arrays to one-dimensional arrays, for memory alignment issues. In the transfer of data between host and kernel, i.e. in the copy from/to the global DDR memory, Xilinx uses a peculiar data alignment, which must be appropriately taken into consideration. It was difficult to correctly align the 2D arrays, which were brought simply to 1D arrays to overcome this struggle. The policy chosen was the row-major so that the consecutive elements of a row reside next to each other. In this way, accessing a the matrix element $[n][m]$ is equivalent to access to the array element $[n \times numColumns + m]$. The code was modified in the points of interest.

The result of this first adjustment is shown below. These are estimates calculated by the Vivado HLS tool in what is referred to as a *synthesis report*. It is a theoretic accurate estimate of the overall - and also partial - number of clock cycles that will make up the final latency, and about the resources occupied by the hardware design.

The Xilinx Virtex® UltraScale+™ VU9P FPGA platform, that is, to the most powerful and capacious among those mentioned above, will be taken as the reference one. The reason is that this model represents the final target for execution. As will be seen, the final application will be run on this card thanks to an Amazon EC2 F1 instance.

An in-depth analysis on this will be made later.

clock	**target**	**estimated**	**uncertainty**
ap_clock	4.0	2.920	1.08

Table 4.4: Timing (ns)

As reported in Table 4.4, it is worth noting that the estimated clock period (nanoseconds) is below the target, i.e., how the clock frequency is higher than the minimum expected frequency. That is a positive indicator. If, on the contrary, the estimated period had exceeded the target period, this would have meant that the sum of the delays related to the operations of the critical path would have been higher than the clock cycle in which a result should be given. The critical path is the longest path within the design in terms of delay.

The uncertainty is because Vivado HLS uses internal models to estimate the delay of each operation. That cannot take into account in advance all the possible increases or decreases of these delays in the real RTL synthesis, with subsequent place and route, unless there is a margin of error. This is clarified in [36, page 29].

Table 4.5 shows the latency for each design cycle in terms of clock cycles. From left to right:

- Loop Name: this is why labels have been placed before the cycles in the

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop_ txAntLoop	190890675776	2982666809	-	-	64
+ coefLoop	431	344	4	1	23
+ symbolLoop	2982666240	24273	-	-	122880
++ interLoop	23437	1019	-	-	23

Table 4.5: Latency of each loop (clock cycles)

code, a practice repeatedly mentioned in section 4.3. The two external loops *rxAntLoop* and *txAntLoop* are executed as one only, since there is no intermediate instruction between the two. Vivado HLS merged the two cycles into one.

- Latency (Max): this is the maximum latency relative to the loop in terms of clock cycles. The minimum latency is not reported because not considered as significant. This can be found in the full synthesis report generated by Vivado HLS. For each estimate, the worst case (so the max latency value) will always be the reference one. The latency related to *rxAntLoop_txAntLoop* represents the overall latency of the kernel, being this the most external loop. The number incorporates the latency of the innermost loop.

- Iteration Latency: this is the number of clock cycles used to perform a single iteration of the corresponding loop. That is multiplied by the trip count to obtain the maximum latency.

- Initiation Interval: this is a fundamental parameter in the concept of pipelining. Usually abbreviated with *II*, it represents the number of cycles necessary for the pipeline to be able to take a new input and process it. The default target is 1: an $II = 1$ means that the pipeline can be fed by a new input at each clock cycle. That represents perfection in terms of execution. Every resource within the pipeline will always and continuously be used without any waste of clock cycles. For waste, it is meant the failure to use a resource for one or more cycles. If there is no number, but the character - means that the loop is not pipelined but executed regularly.

- Trip Count: it represents the number of times each loop is executed. The number refers exclusively to the corresponding loop; this means that, e.g., *coefLoop* iterates 23 times per se, but this will be multiplied by 64, which is the number of iterations of the outermost loop which contains it.

The heaviest loop in terms of overall cycles is *symbolLoop*. The latency of a single iteration is 24273 cycles; this is the number of clock cycles that the device is likely to use for the execution of the single loop. This number is multiplied by the number of iterations, 122880, and again by 64, giving rise to a very high total number of iterations and a consequent burdensome latency in terms of performance.

Multiplying the number of total cycles by the target clock period, the result is $190890675777 \times 4.0 = 763.56s$. It is a considerable time in seconds, about $150\times$ slower than the time taken by the original version executed on the first CPU (average time of $5s$), and not less than $50\times$ slower compared to the software execution on the second CPU ($14.5s$).

It should be noted that the target clock period will be used as a reference, here as below, and not the estimated one. That is because the real hardware platform will perform the operations at the frequency of 250MHz, corresponding therefore to a period of $4ns$.

However, the estimated period will be reported as it must always be below the target period. The reason is that the estimated period represents, with a given uncertainty, the clock period necessary to interface correctly the logic of the critical path. This must be below the indicated threshold ($4ns$) to satisfy the time constraint. In this case, a clock period of $2.920ns$ means being able to carry out the operations at a maximum frequency of 340MHz without issuing timing faults (think for example of the setup and hold times of a FF). So think as the estimated period as an indicator of the validity of the current design, since the frequency of the kernel will still be 250MHz.

The result expected from a hardware accelerator is different. That testifies that using the synthesis tool alone, without deepening specific optimization issues and appropriate test methods, does not lead anywhere.

For what concerns the resources used, the percentage is shown in Table 4.6.

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Total	4	59	22074	16205	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	0	0	1	0

Table 4.6: Utilization estimates

The numbers always refer to the Virtex® UltraScale+™ VU9P platform. These are low numbers as expected, close to zero. Resources are not occupied at all, and the final latency is very high, given the failure to exploit the potential of the FPGA module.

BRAM memory is not used. All the data accessed resides in the global DDR memory, with high access latency and a limited number of ports.

DSPs are also not used; FFs and LUTs are not too. URAM is not used now and will not be used later. This type of memory is not necessary for the purpose.

In the next section, the choices made will be explained step by step and how substantial improvements compared to this initial version have been obtained.

4.6.2 Use of manual pipelining/unrolling

First, the possibility of manually managing the pipelining and unrolling of the loops was assessed. As reported in [37], the XOCC compiler (Xilinx OpenCL Compiler) automatically pipelines loops with a trip count higher than 64, and automatically unroll loops with a trip count less than 64, if there is a real possibility of gaining an advantage. Always according to the guide, this would lead to good results.

In fact, for the final version of the code, this option will be the compiler default, as it will prove to be optimal after various attempts.

Going into this context, in the choice of exploiting the manual pipelining and unrolling commands, the possibility to improve the code shown so far, i.e., the one without any optimization, has been evaluated.

It is indeed correct that the two outermost loops, *rxAntLoop* and *txAntLoop*, merged together, are not pipelined. To understand why these are not pipelined, it is easier to start from the analysis of the innermost loops.

As previously mentioned, the heaviest loop for total execution cycles is *symbolLoop*. Now, this loop is not pipelined: this means that 24237 cycles are used for a single iteration.

It is necessary to force somehow the compiler to use the pipeline, but now it is not possible. The reason is due to many dependencies between the resources accessed inside the loop. The achievable initiation interval would be too high (remember that the ideal one is 1). XOCC attempts various combinations up to $II = 256$, a threshold beyond which it considers the loop as unsuitable for pipelining. In particular, it is precisely the synthesizer to suggest how the problem lies in the dependence between reads and writes on the same bus that leads to global memory *gmem*. The $II = 4$ of *coefLoop* is also due to these dependencies, simultaneous access to resources in memory.

The only slight improvement, not so useful for saving clock cycles but conceptually relevant, is to do the total unrolling of the fourth and final loop, *interLoop*, within *symbolLoop*.

The results are shown in Table 4.7.

As can be seen from the numbers, therefore, there is a saving of $24273 - 23973 = 300$ clock cycles in the critical loop. These 300 cycles saved (for the worst case) must be multiplied by 122880 and again by 64 to obtain the total saving of $300 \times 122880 \times 64 = 2359296000$ clock cycles. Remember that these are just estimates.

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop_ txAntLoop	188531379776	2945802809	-	-	64
+ coefLoop	431	344	4	1	23
+ symbolLoop	2945802240	23973	-	-	122880

Table 4.7: Latency of each loop (clock cycles)

The number seems to tell of a significant saving. It is not a small number in itself, but in this case, it is, if compared to the total overall latency. The expectation is to accelerate the CPU execution several times, while in this case it remains orders of magnitude below.

Furthermore, the estimated clock frequency also drops considerably (see Table 4.8), being still above the target.

clock	**target**	**estimated**	**uncertainty**
ap_clock	4.0	3.633	1.08

Table 4.8: Timing (ns)

The percentage of the resources used is then shown in Table 4.9. The reference FPGA platform is the Xilinx Virtex® UltraScale+™ VU9P (the previous table also referred to it). That will remain the default target platform for all the discussions unless adequately specified.

Please note how the manual unrolling of the innermost loop has led to the greater use of resources, as expected. The improvement in terms of latency is paid with higher resource usage. Right, as these resources are fully available.

The next step will try to solve the problem of accessing the global DRAM memory, copying the resources accessed most frequently in the local BRAM memory.

4.6.3 Use of BRAM

This paragraph reports the results obtained from the use of local memory or BRAM. This memory is vast enough to contain the notable arrays of data, for the current application, in order to obtain a positive result in terms of latency gain.

Remember that BRAM stands for Block RAM, a memory characterized by a faster access time than the global DDR memory, through which the data from the host to the kernel transits. That is a collection of RAM blocks (configurable up to a maximum of two access ports each) instantiated within the FPGA. The dual-port

	BRAM_18K	DSP48E	FF	LUT	URAM
Total	4	124	102941	56081	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	1	4	4	0

Table 4.9: Utilization estimates

nature allows the access to two different locations at the same clock cycle.

In any case and thanks to the OpenCL code, this BRAM can implement different types of memory, e.g., a standard RAM, a ROM, or a FIFO buffer.

Once it became clear that accessing the global memory represented the most severe and significant bottleneck, the solution was immediately resorting to the BRAM. The operation was not so trivial due to the complex architecture of the current code. In fact, this includes the arrays of symbols input to the channel simulation, the coefficients calculated previously, some support structures such as buffers, and many more.

First, an analysis about the size of these arrays was carried out. A second step consisted of the careful evaluation on which arrays were read-only and write-only, or which were both read and written. That led to conscious choices and the minimum risk to face inconsistencies, confirmed by the functional analysis carried out after the changes. Inconsistencies are meant in terms of results, i.e., the symbols exiting the simulation.

As for the transfer of data between host and kernel, the amount transferred is around 60MB, so 480Mb. In section 4.5 was mentioned that the BRAM could contain a total of 75Mb. This number represents only 15% of what is transferred to the kernel and which is actually used by the simulation.

In order to obtain excellent improvements, it was concluded with a repetitive analysis and experiments that there was no need to carry everything in local memory, also because this was impossible. It was enough to transfer 30Mb of that 480Mb (6.25%) in total to obtain a jump ahead in performance. This will be confirmed by the resource usage table that will be analyzed below.

This small percentage reflects in some way the principle of access location. Only these data are accessed more frequently in the code at each iteration, while the others are accessed only sporadically or at most once each iteration. It is, hence, urgent to transport the most accessed resources locally to lower the cost of each access significantly and with it the total cost, gaining in latency.

Faster access means allowing the synthesizer to take full advantage of the pipeline that previously could not be used, precisely because of the too long access times.

The estimated clock period, reported in Table 4.10, increases compared to the case in which resources are not used. That is because the critical path, i.e., the

clock	target	estimated	uncertainty
ap_clock	4.0	3.187	1.08

Table 4.10: Timing (ns)

crossing of the physical resources of the FPGA, is substantially longer than before when everything was read from the global DDR memory. Cases like this one require more complex mapping due to the access to the local memory.

From this section onwards, a further table will be presented, consisting of a single column which shows the total number of clock cycles used by the kernel to carry out the entire task. This amount is the sum of the cycles used for the internal copy of data from global to local memory (and vice versa) and of the cycles used carrying out the normal execution.

The cycles of copy in local memory will not be reported in the cycle table because they are not so fundamental, and since the sum of them turns out to be several orders of magnitude lower than the cycles for the calculation activities carried out by the kernel.

The total cycles no longer correspond to the cycles indicated alongside *rxAntLoop*: the latter remains the outermost loop of the computation part, but these cycles must be added to those of copying the data to BRAM and back.

Latency (max)
6260507146

Table 4.11: Latency (clock cycles)

Multiplying the number of cycles reported in Table 4.11 by the target clock period, this results in $6260507146 \times 4ns = 25.04s$, a time still about $4\times$ longer than that of the first CPU version. That testifies that the effort employed so far is not yet sufficient.

The two outermost loops, *rxAntLoop* and *txAntLoop*, are still merged. Their execution is not pipelined because they contain other loops inside, and therefore it is not possible to pipeline the execution. Dealing with the internal loops of the design, it is known that *coefLoop* passes from an II of 4 to an II of 1, the maximum achievable in terms of performance (minimum latency).

This means that, at each clock cycle, the pipeline is ready to accept a new input, i.e., new data to be processed. That allows the maximum exploitation of resources (and performances), with zero wasted cycles. For wasted cycles, it is meant the clock cycles in which the resources dedicated to the pipeline remain unused.

This is a significant achievement from the more theoretical than the practical point of view. It has been repeated several times (and also clearly visible from

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop_ txAntLoop	6260005632	97812588	-	-	64
+ coefLoop	96	75	1	1	23
+ symbolLoop	97812480	796	-	-	122880
++ interLoop	640	289	16	1	23

Table 4.12: Latency of each loop (clock cycles)

the report) how the latency to be drastically cut is related to *symbolLoop*, which remains unpipelined here.

What are the reasons why it is not pipelined? An answer may be the following: *interLoop*, the internal loop of *symbolLoop*, is not automatically *unrolled*. Even if one tries manually with the directives used in the previous paragraph, nothing changes. The synthesizer is unable to unroll it, i.e., to parallelize it due to memory access problems. By this is meant the multiple access to more than two memory locations at the same time, which is the maximum number for concurrent access to a dual-port RAM.

In the next paragraph, the discussion will cover this problem, working on how to access these resources in parallel at any time.

Going back to the results in the Table 4.12, *interLoop* is executed in pipeline with a II of 16, not good for the small trip count of 23 related to it. Also, in this case, the pipeline leads to a very slight improvement.

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Total	1856	59	49728	70192	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	42	0	2	5	0

Table 4.13: Utilization estimates

Referring to the numbers in Table 4.13, a different scenario is drawn from these concerning the previous cases.

The BRAM, totally unused until now, it is half occupied. Referring to the numbers reported in section 4.1, the arrays in local memory should occupy 30Mb of the 75Mb available, an approximate 40%. Consequently, these numbers are consistent.

The percentage of DSPs, as well as the one of FFs, drops. The percentage of LUTs, instead, rises by a few points. This strongly depends on the automated

procedure by the synthesizer.

The use of URAM remains zero because it is not necessary in this case. Remember that it is an additional RAM that can be used for an even larger number of total bytes; this memory is present only in the models of Xilinx Virtex® UltraScale+™ .

The activity carried out so far has not yet led to convincing results. It has not come to even the CPU time, the first goal of these optimization steps. In the next paragraph, some other techniques will be used to get closer to the final result.

4.6.4 Use of complete array partitioning

The goal remains to ensure that the most demanding loop in terms of execution cycles, *symbolLoop*, is executed in a pipeline fashion. In this way, a fundamental feature of the FPGA would be exploited: the use of the available resources. Latency would consequently drop drastically.

At present, *symbolLoop* is not pipelined. The problem lies in the access to some data implemented using BRAM blocks as a dual-port RAM. It is not possible, in a given clock cycle, to access more than two memory locations at the same time. That severely limits the potential of the application.

It was decided, since, to overcome this problem by adopting the array partitioning technique. Through an OpenCL attribute, extended to Xilinx platforms, it is possible to divide an array following different logics. The most appropriate logic is, in this case, a complete partitioning: each element of the array is mapped so that it can be accessed independently and in any clock cycle. The mapping of every single data takes place using its own BRAM block, or an FF/LUT based on the choice of the synthesizer.

There is a limit, however, and for the current application that is stringent. An array in a OpenCL or C kernel cannot be completely partitioned if the number of elements exceeds 1024. This limitation is imposed by Vivado for reasons related to the use of resources, in particular the use of the BRAM, not allowing to go beyond. This value had no way of being modified with the resetting of a parameter.

Only five arrays in our case are below this threshold, so this technique has only been applied to these. With minimal modification, in general, this method can lead to excellent improvements; unfortunately, this was not the case. Where possible, array partitioning for the current application seems unnecessary.

The five modified arrays are not responsible for bringing complications and consistently increase latency. Indeed, these seem to be of secondary importance in the total calculation. Either way, this change will be kept in the code from now on. In the following work, this method will be repeated on some new array, and the result significant, contrary to what it was seen so far.

As confirmed by Table 4.16 showing the results of the HLS synthesis report, a negligible improvement in overall latency is obtained, which is not sufficient to get

even close to expectations. This improvement derives from lowering the iteration latency of the fourth loop (they would be five, but *rxAntLoop* and *txAntLoop* are counted as one) in the table, *interLoop*, and not from the hoped pipelined execution of *symbolLoop*. It is *interLoop* to gain about one hundred cycles at each iteration, thus leading to this minimal improvement.

clock	target	estimated	uncertainty
ap_clock	4.0	3.183	1.08

Table 4.14: Timing (ns)

The estimated clock frequency has not substantially changed from the previous case. It is still around $3.2ns$, so at a frequency of just over 310MHz.

Latency (max)
5576311175

Table 4.15: Latency (clock cycles)

The total estimated latency is 5576311175 cycles; multiplied by the $4ns$ of the target clock period, it results in $5576311175 \times 4ns = 22.31s$. The result of the software has not been reached yet, still being above the CPU execution time.

In Table 4.17, relating to the resources employed, there is a small reduction in the percentage of BRAM used - 41% compared to the previous 42% - and an equally slight increase in FFs and LUTs for the mapping of the arrays. It does not increase the percentage points.

Once again, it is underlined how generally the adopted strategy can lead to useful improvements. That is proven by the previous thesis work mentioned at the beginning of this chapter. Unfortunately, the current case does not help to appreciate its effectiveness. It also highlights the complexity of the application attempted to accelerate, with results far to be positive.

Having also been this attempt vain, the only reasonable thing was to move on code intervention, left intact so far in all its passages.

4.6.5 Use of pre-calculation

The first real significant improvement occurred with the code changes shown below. It was understood, after various attempts, that the movement of resources in the available memory areas would not have benefited further.

At this point, all the code mechanisms were studied in-depth. That gave space to contain changes that made it possible to evaluate step by step if they benefited and if they were appropriate or not. Even for these modifications, therefore, it was

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop_ txAntLoop	5575809792	87122028	-	-	64
+ coefLoop	96	75	1	1	23
+ symbolLoop	87121920	709	-	-	122880
++ interLoop	553	553	16	1	23

Table 4.16: Latency of each loop (clock cycles)

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Total	1806	55	60386	71027	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	41	0	2	6	0

Table 4.17: Utilization estimates

chosen to proceed by attempts. In a certain sense, it was necessary for diverse reasons.

In the context of high-level synthesis, various automated phases are used, as specified in section 3.1. In the automatic synthesis phase, what at first glance can be labeled as a good idea can prove to be a failure. That is because the synthesis is carried out by a tool that is often not exceptionally performing, where the user does not have only a minimal control option. As in the case of a hardware description language, there is no full control of resources and states of what is performed. In this way, the lack of control is the price to pay to cut design time.

The changes made, substantially due to the lowering of the overall latency, relied on a simple idea: the preventive loading of some resources frequently accessed in the BRAM memory. In this regard, a study has been carried out on the most accessed resources during the critical kernel cycles.

From this, it emerged that some critical data were read and overwritten (often with accumulations) repeatedly in BRAM arrays, structures larger than 1024 elements, and therefore not partitioned to registers. It was then clear that there was an exact location of the accesses, being the purely iterative calculation and based on known indices, which allowed to know in advance the exact locations accessed.

Therefore, other smaller arrays (with less than 1024 elements) that could be implemented with registers, filled with frequently accessed data, were declared.

For clarity, some example code is shown below, which greatly simplifies the

original code. With this, the changes made can be seen.

```
double arrayBRAM[N*M];   // array in BRAM with N*M > 1024

Loop1: for (int i = 0; i < N; ++i) {
    Loop2: for (int j = 0; j < M; ++j) {
        do_something(arrayBRAM[i*M + j]);
    }
}
```

```
double arrayBRAM[N*M];   // array mapped as a unique dual-port BRAM
    block with N*M > 1024
double arrayPart[M]__attribute__((xcl_array_partition(complete, 1)));
       // completely partitioned array with M <= 1024


Loop1: for (int i = 0; i < N; ++i) {
    copyIntoRegs: for (int l = 0; l < M; ++l) {
        arrayPart[l] = arrayBRAM[i*M + l];
    }

    Loop2: for (int j = 0; j < M; ++j) {
        do_something(arrayPart[j]);
    }
}
```

The concept is trivial. In the first code shown, the program wants to access a bidimensional array of $N \times M$, with N number of rows and M number of columns. It accesses the elements of the first row, then in the next iteration the elements of the second row and so on. Since the number of elements $N \times M$ is above 1024, as indicated in the commented code, it is impossible with an OpenCL attribute to completely partition the array. The *do_something* function must, accordingly, access the array in BRAM being able to take only two data per clock cycle, not allowing the full unrolling (that is, complete parallelization) of the loop. To make that possible, each element of the Nth array row should be accessed at any time.

Before *Loop2*, the first row of array in BRAM was copied in the partitioned array (arrayPart); in the next iteration, the second line will replace the first, and so on. It is evident that, in this way, an overhead is added due to the new copy cycles, but this is not very important compared to the gain of latency obtained.

Please note that *do_something* is a fictitious function; in this case, it is assumed that it "does something" by reading the data and not overwriting it. If a rewrite

occurs, in fact, a further cycle should be added at this point following *Loop2*, which provides for the reverse copy, that is, from array part to array beam. Even this overhead can be considered absolutely secondary in a broader context than the current one, where advantages are always significant.

The change reported has been accompanied by other minor changes that will not be mentioned because they are out of context. The tables of the HLS summary will now be reviewed to appreciate the improvements of this version.

clock	**target**	**estimated**	**uncertainty**
ap_clock	4.0	3.569	1.08

Table 4.18: Timing (ns)

The estimated clock period grows, which means a lower clock frequency.

Latency (max)
196903095

Table 4.19: Latency (clock cycles)

As reported by the latency estimation, instead, the overall maximum drops to 196903095 cycles, a much lower number than the initial one. Compared to the non-optimized design, using 190890675777 cycles to complete its execution, the current number means having made an improvement of $190890675777 \div 196903095 = 970\times$ approximately. Compared to the case seen just before, however, with its 5576311175 total cycles, the improvement is $28\times$.

The estimated latency, if multiplied by the target clock period of $4ns$, gives an execution time of $196903095 \times 4ns = 0{,}788s$. The result is now positive, being a time of $5 \div 0.788 = 6\times$ less. However, the result still does not represent the lowering of latency by one order of magnitude ($10\times$), the minimum target desired and expected.

Moving on to Table 4.24 you can immediately notice where the improvement comes from.

Finally, *symbolLoop* is pipelined. The II is still high (25), so other improvements related to this are expected. The key to a further drop of latency lies here, in lowering the II as much as possible to make it close to the minimum, i.e., near 1. 25 is a high number. The execution waits for 25 sequential clock cycles between the process of one input data and the next one, effectively wasting 24 clock cycles. It waits for an operation to be completed without being able to do anything in the meantime. In other words, all the reserved resources in the pipeline are not used for 24 cycles.

In any case, this result is undoubtedly an excellent point to start wanting to

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop_ txAntLoop	196647744	3072621	-	-	64
+ copy	24	2	1	1	23
+ coefLoop	95	74	1	1	23
+ symbolLoop	3072461	487	25	1	122880
+ save	23	1	1	1	23

Table 4.20: Latency of each loop (clock cycles)

improve further.

Compared to the previous section, it is worth noting that two cycles have been added and one removed from the table. In fact, *copy* and *save* were added: these carry out the substantial modification shown in the example above.

In this case, 23 elements are copied first from BRAM to redundant smaller arrays completely partitioned; then, the same 23 updated elements are copied inversely to BRAM. The overhead is negligible compared to the overall number of cycles and the *symbolLoop* cycles, as repeatedly expressed previously.

clusterLoop2 no longer appears inside *symbolLoop*, as it is now completely parallelized thanks to the new changes. The synthesizer unrolls it by default. The memory accesses that previously were an obstacle have been removed. That is what allowed the first run of *symbolLoop* in a pipelined fashion.

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Total	1974	69	96815	68104	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	45	1	4	5	0

Table 4.21: Utilization estimates

As regards the use of FPGA resources, as indicated in Table 4.25, there is an increase of a few percentage points of BRAM and FF, but nothing too evident or burdensome. It is quite logical that the more the kernel is optimized, the more the resources are exploited.

4.6.6 Modification of the code: algorithm

As a second optimization, it was chosen to intervene directly on the algorithm by minimally modifying its operation. As will be seen, this change will lead to another

significant improvement in performance.

According to the TIM supervisors, this change represents the code as it was in its original form, then modified by them and made more generic without there being a performance gap between the two versions executed purely on CPU.

On the FPGA, however, this gap exists and is consistent (at least from the estimations), clearly visible from the tests made. Not wanting to confuse thoughts by bringing back the code in its entirety, only part of the code is extrapolated from the broader context and reported below as a simplified version, as it was also done in the previous section.

One of the problems found responsible for a high initiation interval (a minimum of 25 has now been reached) is the way an accumulation was carried out within *symbolLoop*. A constant was added to a variable at each cycle; this constant came from a fully partitioned array, accessed through an index retrieved by a large array in the BRAM memory. By studying the content of this last array, it was precisely understood that the contained indices were all the same if a parameter of the simulation design was constant.

It was therefore decided to move from a first general case, variable according to the mentioned parameter, to a more specific case. It was also verified that by changing the parameter to maintain the simulation goodness, the performance remained more or less the same.

In other words, the code from this point on is valid if a specific simulation parameter is N. If this parameter is changed to M, the code must be rearranged to follow its modification, but without performance penalty in execution.

However, the transition from a general form to this single case is plausible since the software execution times were calculated when the mentioned parameter was N, that is, in the whole harmony with the change made. More specifically, software simulation times (on CPU) increase with the increase of this parameter. In the case of hardware, with the current modification, this behavior would not occur, at least regarding the fragment of code that will be shown, or the accumulation.

If increased a lot, this parameter could be problematic in some array declarations; this is because the parameter directly controls the size of these arrays. If too large, some arrays may no longer fit properly in the BRAM or be partitioned, and this would imply a possible slowdown in execution.

A more in-depth study about this eventuality is not reported because it was considered outside the scope of the discussion. The appropriate assessments were made with the TIM supervisors.

The simplified code will now be briefly reported. Accumulation occurs starting from an initial value to which a constant is added to each cycle, as mentioned above. The initial value is A, while the constant is B. The subscripts refer to the number of the iteration.

$A_0 = A_{init} + B;$
$A_1 = A_0 + B;$
[...]
$A_N = A_{N-1} + B;$

That is what happened in the code. The constant B was added to the initial value of A, A_{init}. It occurred in a more sophisticated way since B was found by a vector accessed indirectly through an index found by another vector, as mentioned before. In any case, the code shown makes the idea of the simple accumulation carried out.

The problem is that accumulation is a multi-cycle operation. It requires the reading of the first operand A, the reading of the second operand B, the sum of the two operands (floating-point sum), and the rewriting of the result in one of the locations read previously.

This means that if, for example, the operation takes 12 clock cycles, the next operation carried out in the next iteration should wait for 12 clock cycles, stalling. This is because A_{K-1} would be needed for the calculation of A_K (with $k > 0$), which would be calculated in 12 clock cycles.

That could not guarantee to drop below a 12-cycle initiation interval, theoretically. This means not being able to pipeline the accumulations. The synthesizer should, in principle, understand that it can get around the obstacle. However, it is not able in the case of the original code since this constant is found by the complex mechanism of multiple addresses mentioned above.

The code was changed by hand to make sure that the obstacle is overcome and the wait is canceled. A simple replacement is used in other to do that.

$A_0 = A_{init} + B;$
$A_1 = A_0 + B = A_{init} + B + B = A_{init} + 2*B;$
[...]
$A_N = A_{init} + (N+1)*B;$

In this direct and straightforward way, the dependence of A from its previous iteration value is removed. Using other words, A_K no longer depends directly on A_{K-1}. Hence, it will no longer have to wait for its actual completion. A will now depend only on A_{init}, the initial value of A contained in a register accessible in any iteration.

Note how the change becomes possible thanks to the fact that B is a constant. If this were not the case, there would be no way to get around the difficulty if failing to predict in advance the value that will be added, but this requires considerable effort, and it is not always possible to reach a hoped-for conclusion.

In addition to this change, to be considered as the main one, a second modification was made, which allowed a considerable saving of occupied BRAM. That is how

the vast array of outgoing symbols is used. The latter is filled by the kernel and passed from device to host: these are the outgoing simulation symbols. Until now it had been treated as a read and write buffer, for lack of attention. The approach to it has changed in one of the final stages of finishing, and this has brought a significant additional advantage.

It was understood that the reading took place in two sections. In the first half of the simulation, only the first half of the array was filled. Consequently, in the second half of the simulation, only the second half of the array was filled. This fact allowed us to reserve a local array in BRAM for the storage of these symbols. An array in BRAM is essential because of faster access since these symbols are the object of operations like accumulations. Hence, these must be written and read quickly. That was done by halving the overall size of the array, the two halves being written in separate times. Obviously, in the middle of the simulation, this local array is saved in the global buffer to be then sent to the host without losing the symbols computed in the previous half iterations.

The combination of these two changes resulted in a significant upgrade to the code and estimated run time. Remember that these are estimates by the high-level synthesis tool that should, in any case, come close to the truth.

In the reports listed below, it can be seen how the pipeline is now better exploited, leading to a rather significant number of cycles; the accumulation operation is now well parallelized. The discussion also focuses on the use of resources, where it is clear how a substantial part of local memory has been saved.

clock	**target**	**estimated**	**uncertainty**
ap_clock	4.0	3.187	1.08

Table 4.22: Timing (ns)

As for the clock period, the estimate remains fixed at $3.187ns$. Hence, the frequency is now 313.775MHz, and the time constraint is still met.

Latency (max)
102409471

Table 4.23: Latency (clock cycles)

The overall latency now settles on the 9-digit limit. The estimated execution time is $102409471 \times 4ns = 0.41s$. The result is positive being the execution time, or the number of cycles represented by the latency halved compared to what was shown in the previous section.

First, note how the two outermost loops, so far considered as a single loop, are now separate. The tool merged them into one since there were no instructions

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop	102400148	51200074	-	-	2
+ txAntLoop	51138496	1598078	-	-	32
++ copy	23	2	1	1	23
++ coefLoop	95	95	1	1	23
++ symbolLoop	1597919	493	13	1	122880
++ save	23	1	1	1	23
+ outSymb	61574	139	4	1	15360

Table 4.24: Latency of each loop (clock cycles)

interposed between the two mentioned loops. Now, instead, the copy cycle of the *outSymb* symbols is in between, and the two outermost loops are separated.

This is only for the record as the fact is not responsible for any improvement or deterioration.

Much more important, however, is the passage of the *symbolLoop* initiation interval from 25 to 13 clock cycles. That is what allows the final latency of the simulation to be nearly halved, always according to the estimates. It has been repeatedly stated that the total latency calculation is linked to this factor. That is because *symbolLoop* is the cycle that performs the work of computing the output symbols.

For this reason, halving the II of *symbolLoop* means halving the total latency. This achievement, therefore, allows us to settle on the theoretical $0.3s$, reaching a speedup of almost $20\times$ compared to the best CPU simulation time.

As can be seen, finally, the need to save the symbols in the global array (that will be transferred back to the host at the end of the work) is added to the total calculation only twice in the entire execution of the kernel code. The reference loop is *outSymb*. Being that loop performed only twice, the contribution in terms of cycles is not to be considered significant. Moreover, the copy of these values from a vector in BRAM to one in global memory takes place in a burst. Hence, the total cycles necessary for copying all the symbols decreases by $8\times$ (taking full advantage of the bus for the transfer). The aspect of the burst transfer is significant. It will be addressed later, especially in the final part of the chapter.

Regarding resource usage, the main focus is on the first item, namely the use of BRAM. As anticipated, the second modification of the kernel leads to a fairly substantial decrease in the use of it, reserving space for half an array of outgoing symbols instead of for the entire array as previously happened. That means that the percentage settles at around a quarter (26%) of the total available.

	BRAM_18K	DSP48E	FF	LUT	URAM
Total	1128	154	115859	80140	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	26	2	4	6	0

Table 4.25: Utilization estimates

The DSP usage also increases from about zero use to 2%. Several operations are therefore parallelized, and the DSPs, complex logical blocks, used following the new calculation needs.

The data relating to FFs and LUTs remain instead mostly unchanged.

The changes reported gave a positive result to the efforts put in place to achieve the expected result. It is fair to mention for the thesis, not as a justification for the work done, that the unexpected events were many and that the choices made and reported came only after numerous unsuccessful attempts. For fluidity, these attempts are not mostly reported, except in some cases considered necessary. In fact, during the writing, it was clear that by proceeding in some sensible ways at first glance, the desired result was not obtained.

During the following section, a further and final modification will be considered. That will allow the achievement of the optimization initially expected. That concerns a modification to the code that derives from a more in-depth analysis of the hardware generated automatically by the synthesis tool.

4.6.7 Modification of the code: memory

The changes that will be reported later derive from a careful analysis of various reports generated by the Vivado synthesis tool with the help of the team under the professor's Lavagno supervision. The goal was to drop below the Initiation Interval of 13 for the innermost loop *symbolLoop*. The compiler gave suggestions that proved to be correct.

The problem lays in a buffer that is first written in a specific location, then read at a location that could be the same or different, according to a more complex indexing mechanism. This buffer is a simple 1600 element array of doubles. If there was the certainty that in specific cycles, and with a fixed frequency, this buffer is read and written at the same location, then the code could be changed by replacing the reading of the buffer location with the reading of the value that is written in the buffer. However, the latter being read from global memory, the success of the operation would remain doubtful.

A scheme will be reported to represent the access problem, first reporting the reduced and simplified C code.

```
symbolLoop:
for (unsigned int i = 0; i < 122880; ++i) {
    int A = something(...);
    buffer[A] = data;

    interLoop:
    for (unsigned int j = 0; j < 23; ++j) {
        int B = somethingElse(...); // A and B can be equal or different
        out = buffer[B];
    }
}
```

As can be seen easily, within *symbolLoop*, the buffer is accessed in writing at location A. This index is found by a fictitious function, in this case, indicating that it is the result of multiple passes and not at all obtained in an easily interpretable way.

Within *interLoop*, the same buffer is accessed for reading to a location that may differ or be equal to that of the previous write, as reported in the code comments. In fact, the access index is this time B, found in a similar way to A, but through other passages.

The real problem is *interLoop*. In fact, in order for *symbolLoop* to be carried out in the pipeline, which is essential and necessary for the performance to achieve, then *interLoop* must be completely unrolled. That means that the tool parallelizes the operations, with the loop that is now removed. The unrolled code turns into something like the code presented following.

```
symbolLoop:
for (unsigned int i = 0; i < 122880; ++i) {
    int A = something(...);
    buffer[A] = data;

    int B0 = somethingElse(...);
    out0 = buffer[B0];

    int B1 = somethingElse(...);
    out1 = buffer[B1];

    [...]

    int B22 = somethingElse(...);
    out22 = buffer[B22];
}
```

From here, the problem is noticeable. The buffer in memory is accessed once in writing, then 23 times in reading within a single iteration, at which the II rises to 13. The RAM that implements the buffer is dual-port, so accessible at two different locations in a single clock cycle.

Now, this is written to location A, and this steals an execution cycle (assuming it is one only), and the next 23 readings are taken two by two in 12 cycles. That makes perfect sense and briefly explains the problem faced.

The simplest way to overcome the obstacle would seem to be working on how this buffer is stored, that is, with what resources it is implemented in the FPGA local memory system.

A complete partitioning or reshaping allows to implement an array using a block/register for each element; however, Vivado HLS limits the complete partitioning/reshaping to 1024 elements, as mentioned several times. The buffer is composed of 1600 elements. There is no way to overcome this limit imposed for apparent reasons of space and to ensure with relative safety that it does not go beyond the use of permitted resources.

There is also another method similarly based on how the buffer is being implemented, which is now under discussion. It is about using an HLS *pragma* for manual resource control. That statement is not possible in the OpenCL language, so the kernel was rewritten in C with a few lines of code of difference.

The advantages of a transition to a C/C++ kernel are indicated in more comprehensive detail in subsection 4.6.8. These advantages mostly reside in an extensive control of resources possible thanks to greater compatibility of the Xilinx tools towards C, in fact, compared to OpenCL.

It has been tried to ensure that the buffer was implemented using resources accessible at any time to avoid long waits leading to a II of 13, a large number which indeed represents a performance limit to be overtaken.

According to [36, Table 1-28], it is possible to push Vivado HLS to use LUTs to implement a dual-port asynchronous RAM memory. The resource, or core, in this case, is called RAM_2P_1S. A pragma HLS_RESOURCE was used specifying our buffer as reference data and RAM_2P_1S as reference core.

SW-Emulation and HW-Emulation worked, and the HLS report was encouraging. The problem laid in the system build, i.e., in the generation of the bitstream for programming the actual target device. Vivado HLS did not realize that there could be an actual implementation problem related to the use of resources.

As if nothing had happened, in the table of the resources used the percentages were well below 100%. However, in the attempt to build the system, there was an error indicating that a security parameter (MemorySizeLimit) had been set to a default value.

Shortly, an attempt was made to circumvent the partitioning/reshaping limit by attempting a secondary strategy, but running into the same problem of a memory

threshold too severe for a real case like this.

However, the existence of a Vivado command was suggested in the console to modify this limit if necessary. Instead, there was no way to change the partitioning/reshaping threshold parameter.

The command was entered in the linking phase of the system build. In fact, inside the build process, there are two phases in which XOCC executes different scripts: a first compilation phase (*xocc -c*) and a second linking phase (*xocc -l*).

The parameter was successfully modified in this way, but a subsequent problem appeared, this time not viable by merely changing a threshold. The problem mentioned stemmed precisely from the forced overcoming of this.

There were not enough resources for the required implementation. More LUTs, more FFs than those present and available on the reference FPGA, had to be used for the purpose, which was plausible, but strange since after all the buffer of massive size, but only about 1.5× higher than the limits set default.

On the one hand, the limit placed made perfect sense. Going beyond this threshold meant colliding with the physical limit. On the other hand, some suspicion persisted. It was proceeded with an analysis of the Verilog file generated by Vivado in the synthesis phase. The automatic tool tried to implement the buffer with a 23-port RAM so that, after writing, all 23 readings could take place simultaneously.

The strategy worked in HW-Emulation but again not in the system build. The summary report showed that the resources to be used in order to implement the solution were above the limit.

There is no other pragma directive, or rather a core to be specified within a directive, which implements the buffer in a further convenient way. The only solution seemed once again to be the manual intervention on the code.

The right intuition, however, came thanks to reflections on the buffer implementation inside the RTL Verilog file. Could a user manually and by intervening on the code recreating a sort of a 23-port RAM that contains the full buffer? In other words, is it possible to modify the buffer so that it can be accessed simultaneously from these 23 readings? The answer is yes, by sacrificing memory, that is, by making the buffer redundant.

So, what was done was creating 23 identical copies of the buffer. Each of the 23 parallel reading executions accesses its copy at location B, which will be written to the previous cycle location in location A. In other words, identical writing takes place at location A for all 23 different copies of the buffer. In contrast, the 23 readings take place at location $\mathrm{B_X}$, with X (between 0 and 22) indicating the index of the iteration.

Copies were created by making the buffer a two-dimensional array where the first index accesses the "copy" while the second index the concerned location.

The modification made to the code is shown below so that the advantage can be visible.

```
symbolLoop:
for (unsigned int i = 0; i < 122880; ++i) {
    int A = something(...);

    interLoop:
    for (unsigned int j = 0; j < 23; ++j) {
        buffer[j][A] = data;

        int B = somethingElse(...); // A and B can be equal or
    different
        out = buffer[j][B];
    }
}
```

Once unrolled, the *interLoop* is automatically transformed into the following.

```
symbolLoop:
for (unsigned int i = 0; i < 122880; ++i) {
    int A = something(...);

    int B0 = somethingElse(...);
    buffer[0][A] = data;
    out0 = buffer[0][B0];

    int B1 = somethingElse(...);
    buffer[1][A] = data;
    out1 = buffer[1][B1];

    [...]

    int B22 = somethingElse(...);
    buffer[22][A] = data;
    out22 = buffer[22][B22];
}
```

One detail is still missing. Creating a redundant buffer, as in this way, does not solve the problem: by default, the buffer is implemented in any case with a BRAM block that implements a dual-port RAM, so the original problem persists.

It is to make sure that the 23 copies are implemented using 23 different BRAM blocks, and that each block implements a dual-port RAM so that every copy can be accessed independently.

To do this, just an array partitioning of the array called *cyclic* with a factor of

23 has to be done. In this way, it is as if 23 independent copies of our buffer were created. The code that represents this change could be the following, although this, it should be remembered, is carried out automatically by the synthesis tool following the indication given.

```
symbolLoop:
for (unsigned int i = 0; i < 122880; ++i) {
    int A = something(...);

    int B0 = somethingElse(...);
    buffer0[A] = data;
    out0 = buffer0[B0];

    int B1 = somethingElse(...);
    buffer1[A] = data;
    out1 = buffer1[B1];

    [...]

    int B22 = somethingElse(...);
    buffer22[A] = data;
    out22 = buffer22[B22];
}
```

The result is reasonably convincing in terms of estimated performance, as highlighted by the reports while presenting an overhead of memory usage. The overhead of 23 times the original is a high amount, if taken in itself, but represents an affordable percentage overall, as the reports show.

clock	target	estimated	uncertainty
ap_clock	4.0	3.137	1.08

Table 4.26: Timing (ns)

The estimated clock period drops slightly, settling at $3.137ns$. It is around 320MHz in frequency. The frequency data is always to be considered relatively, as well explained in subsection 4.6.1. As will be seen from the tests carried out on the FPGA, this will drop to 250MHz, i.e., the target frequency indicated in the table ($1/target\ period$).

As for the overall latency data, this drops considerably to $16022905 \times 4ns = 50.264ms = 0.064s$, about $80\times$ faster than the best CPU simulation value. Finally, it comes to an expected and difficult to improve result.

As can be seen from the picture that takes the loops in analysis, now the main

Latency (max)
16022905

Table 4.27: Latency (clock cycles)

	Latency		Initiation Interval		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop	15892798	7946399	-	-	2
+ txAntLoop	7884800	246400	-	-	32
++ copy	23	2	1	1	23
++ coefLoop	95	74	1	1	23
++ symbolLoop	246241	484	2	1	122880
++ save	23	1	1	1	23
+ outSymb	61574	160	4	1	15360

Table 4.28: Latency of each loop (clock cycles)

loop or *symbolLoop* has an $II = 2$, the desired result. The change had the desired success. This factor 2 derives from the code shown above: at present, the buffer is written in one clock cycle and read at the next one. This represents the minimum number of clock cycles between one iteration and the following, so the pipeline will have to wait in fact for two clock strokes for new inputs to be processed.

Without a further in-depth analysis and a probable drastic change of algorithmic choice, in all probability, one cannot think of going below this value, however positive in terms of hardware acceleration, the final aim of the work.

Table 4.29 shows an estimate of the resources used. Note how it goes from 26% in the use of the BRAM to a more substantial 34%. That is due to the overhead introduced following the modification of the buffer. 23 copies of the same buffer were made, and these affect the total computation of the BRAM blocks used.

Other resources are also experiencing an increase. That is the practice, being now the much more performing application. To achieve these latency results, the FPGA maps operations on the most significant number of resources available, making full use of its capabilities.

DSP usage also increases significantly, from a starting 2% to a final 15%. The FFs grow to 11% starting from the initial 4%. LUTs also reach 14%.

The Ultra RAM remains unused because, in these circumstances, additional memory is not needed. The BRAM is sufficient in order to store the requested information. If more space was needed or if the resources used had to be balanced, then it would also have been used. That would have implied switching from OpenCL

	BRAM_18K	DSP48E	FF	LUT	URAM
Total	1484	1026	283636	172742	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	34	15	11	14	0

Table 4.29: Utilization estimates

to C in order to control the URAM, hence to map data to it manually.

This section is the last one relating to the actual optimization of the kernel for the simulation of the channel model. The results shown are the final reference ones, so it can be said that the goal has been achieved.

In the next brief section, the discussion will continue to cover the code optimization. Also, the possible advantages of moving from the OpenCL kernel to a C kernel will be indicated.

4.6.8 Possible benefits of switching to a C/C++ kernel

XOCC accepts C/C++ kernels or OpenCL kernels as input source files. It has already been mentioned that, in the case of a C/C++ kernel, the designer has more control over resources. But not only. The SDAccel™ environment supports the OpenCL C language and the related built-in functions related to the embedded profile OpenCL version 1.0. [29] This version represents a subset of ISO C99 with extensions for parallel calculation.

It should be remembered that the OpenCL framework consists of a set of APIs to manage and support parallel computing in the field of heterogeneous computing and execution on different platforms/processors. However, OpenCL also indicates the programming language (as in this case) with a peculiar development environment [38]

As for the use of OpenCL source files, it is possible to specify (as previously seen) inside them various attributes capable of optimizing system and kernel performance. In general, the XOCC compiler (and with it the SDAccel environment and the SDx software used in the thesis), and the Vivado HLS tool, can interpret these attributes precisely for two specific tasks. The first concerns the so-called data movement, i.e., the optimization of how data is transferred from host to device and vice versa, to maximize the throughput and transfer bandwidth of the DDR memory. The second, as already said and seen, concerns the optimization of the kernel, making sure that the input data is consumed as soon as possible once they arrive at the kernel interface [29].

The thesis work - with the progress made up to here - was based on this second type of optimization. The original code has been expanded and enriched with code to support loop pipelining, unrolling and array partitioning. Data flow techniques

(data transfer between multiple kernels within a design) and function inlining were not considered because they were not strictly needed. The OpenCL attributes used also include additional Xilinx attributes, those that begin with *xcl_*. With source code in C/C++, supported by the same compiler as already stated, the SDAccel™ environment fully supports HLS optimization techniques. These prove to be more numerous than the previous ones with regards to optimizing kernel performance since they can take advantage of all the possibilities offered by HLS being provided only for C/C++ sources. Remember that Vivado HLS is the tool that synthesizes an RTL kernel from sources in OpenCL, C, and C++ languages.

That provides an integrated pragma for design optimization such as reduction of latency and maximization of throughput, control of the resources of the final RTL code. These pragmas are direct to be inserted in the source code and directly interpreted by the synthesis tool.

As regards the control of resources, mentioned several times in the course of the writing, there is a precise HLS pragma for this purpose. Imagine wanting to use some URAM memory blocks (UltraRAM of the UltraScale series) for a more appropriate distribution of resources, for example, in a case such as the one discussed in subsection 4.6.7. It is possible without any effort to map data arrays into URAM using a specific HLS pragma.

Think of having a buffer of size N to be implemented in local memory. It will be mapped to BRAM by default, or URAM if specified #pragma HLS RESOURCE variable = buffer core XPM _MEMORY uram. With dedicated pragmas, it is also possible to decide with which resources a given operation will be carried out: for example, if a DSP or not will be used for a floating-point multiplication. This extensive control of resources, it is reiterated, is possible only in a C/C++ kernel. If such pragmas are inserted in a CL source, however, they will be ignored by the compiler.

The advantages of a C kernel also concern the manual configuration of the communication interface between FPGA and global memory on the development board. The protocol used and supported by the Xilinx boards is AXI4, together with its AXI-Lite version. With the appropriate pragma, it is possible to indicate additional information advanced to the kernel. By default, for example, all interfaces are connected to a single global memory bank called *gmem*. Since the interfaces are configurable, it is possible to map the data coming from the host on different banks, redirecting the interfaces if necessary. In this way, it is possible to further parallelize the movement of data by exploiting more memory banks and more available ports.

That was only a very brief overview of the opportunities offered by fully exploiting the HLS pragma set.

4.6.9 Optimized code on different FPGA models

Now a brief comparison between the three FPGA platforms mentioned in section 4.5 will be presented as regards the final implementation of the code, i.e., the final optimized one. The code so far has been modified using as target the first FPGA platform, which is the one made available on Amazon EC2. The latter cloud computing environment will be briefly presented later. An in-depth study has not been carried out on the other two platforms, neither the code modified appropriately to achieve the maximum hoped results for having one of these two models as the reference model.

The results of the HLS reports are aimed at showing the intrinsic differences of the devices. Different periods (especially between the first Virtex® UltraScale+™ and the third Zynq®), different technologies on silicon, different capacities.

Only the reports relating to the clock frequency and those relating to the resources used will be reported. This is because there are no substantial differences, however, in the latency report.

Kintex® UltraScale™ KCU115 As for the second FPGA model presented previously, the Kintex® UltraScale™ KCU115, the clock frequency remains identical to that with the FPGA model Virtex® UltraScale+™ VU9P.

clock	target	estimated	uncertainty
ap_clock	3.3	3.137	0.90

Table 4.30: Timing (ns)

The target frequency and uncertainty change but will not be discussed as there won't be a chance to try the board physically.

	BRAM_18K	DSP48E	FF	LUT
Total	1488	277	432580	315840
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	34	5	32	47
Utilization SLR (%)	68	10	65	95

Table 4.31: Utilization estimates

In the resource report, Table 4.31, it is noted that the SLR entry, which must be respected, affects the usage to be very high and almost close to the limit. The concept of SLR is briefly reported in section 4.5.

If it were not for the SLR percentage, the total calculation of the resources would be reasonable and distributed fairly. The BRAM usage remains at 34%, as in the other FPGA platform. Note that URAM is not present on this model, as it is the Kintex® UltraScale™ series, and not a the Virtex® UltraScale+™ .

Instead, the percentages of FFs and LUTs tripled. Above all the others, LUT SLR usage rise to 95%, once again demonstrating the magnitude and complexity of the realistic case in question.

The smallest number regards the use of DSPs. One can think of switching to a C/C++ kernel to better map operations, e.g., moving the most recurrent floating-point multiplications over DSPs, leaving LUTs more free. This is only possible through the Vivado HLS pragmas, so not using an OpenCL kernel, as reiterated in the previous section. It seems that the FPGA model described is suitable for containing the optimized code.

Zynq® SoC ZC706 As for the ZC706 board, using an outdated technology, usage reports are very different.

clock	**target**	**estimated**	**uncertainty**
ap_clock	10	7.300	2.70

Table 4.32: Timing (ns)

The estimated clock period is more than double that of the two previous models. The target frequency is quite different, as evident. The uncertainty is also higher. A period of $7.3ns$ roughly corresponds to a clock frequency of 140MHz. Already from here, it is clear how such a model is reserved for different purposes.

	BRAM_18K	**DSP48E**	**FF**	**LUT**
Total	1720	1026	212330	216541
Available	1090	900	437200	218600
Utilization (%)	157	114	48	99

Table 4.33: Utilization estimates

For what concerns the hardware resources, the estimated usage of BRAM goes well beyond the limit. 157%, one and a half times the maximum allowed. BRAM is the only RAM available on the card (no URAM, as in the UltraScale+™ platform). Without a change in the design, it is not possible to overcome this limit. A change in the design would cost a great effort to arrive at unsettling results probably.

The percentage of DSPs in use is also above the threshold. Even the technique of replacing some operations with LUTs instead of DSPs would not work since they

are hitting the limit, even with LUTs.

The use of FF, instead, settles on 48%, the only comforting data.

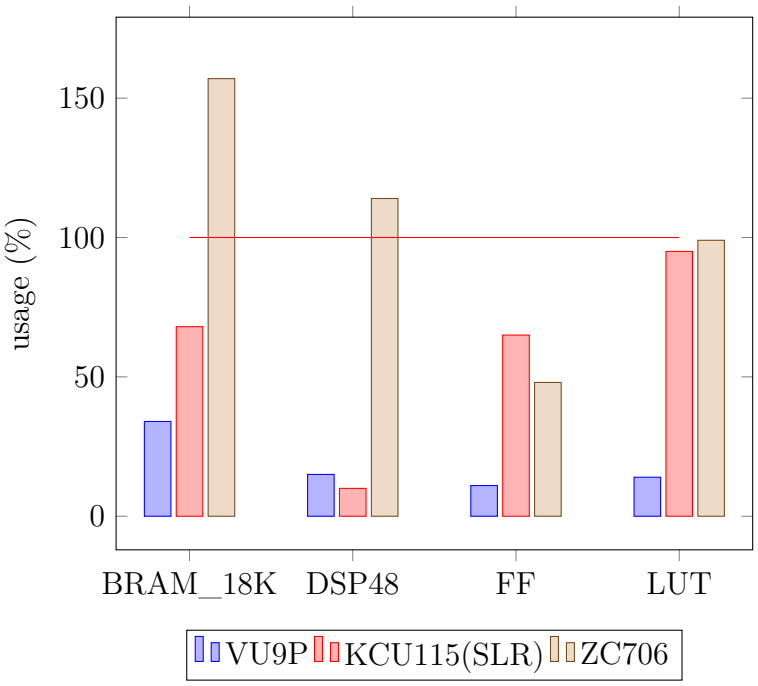


Figure 4.4: Resource utilization on different Xilinx FPGA acceleration boards

Therefore, there is a substantial problem with reaching such optimization levels with a non-new generation FPGA. One solution could be to settle for half of the performance in terms of overall latency by increasing the target initiation interval to 4. A low limit of 2 has been reached; now, it can be raised to 4 obtaining, in theory, the half of the performance (i.e., double latency) with half the resources, in a linear way. That would be true for DSPs, FFs, and LUTs, but not for BRAM. That remains a significant problem.

4.7 Test on Amazon EC2

4.7.1 Short overview

To test the goodness of the results, the AWS cloud platform was leveraged. AWS (Amazon Web Services) offers paid infrastructure and on-demand services for a variety of purposes and functions.

In particular, for that of this thesis, the reference will be the Amazon Elastic Compute Cloud (Amazon EC2), a web service that offers cloud computing capabilities in a scalable and secure way. These allow access to virtual servers in the cloud,

machines physically located in specific areas of the world. To give an example, the data centers relating to the activity that will be discussed are present in the US East (Virginia), US West (Oregon), and Europe (Ireland).

Cloud computing is undoubtedly a solid reality for the majority of companies, from startups to large corporations, precisely because it is possible to rent entire infrastructures, services, or microservices made available by a cloud provider for variable periods. This latter is Amazon, in the following case. Other cloud providers include Microsoft, IBM, Oracle.

The price varies according to the type of infrastructure (and attached storage or calculation capacity) and the period of use. The price is also determined by maintenance costs and the complexity of interfacing between a rented infrastructure and the final user.

In AWS EC2, it is possible to create instances to take advantage of the hardware acceleration of a Xilinx Virtex Ultrascale+ FPGA, the same chosen for the code optimization so far. Other clouds offering FPGA services include Huawei, Alibaba, Baidu, Tencent, and Azure. The service allows verifying the goodness of the application accelerated not only by simulation reports but by real statistics deriving from the execution of code on physical hardware.

These instances are created within AWS EC2 and are called F1 instances. These instances are proper virtual machines that the user accesses through a public IP address, after their creation. Each created instance is called AMI (Amazon Machine Image): it is a Linux image, which presents the development tools, run time, and libraries required to use FPGAs as hardware accelerators. Therefore, an AMI can be used both in development - SDK and HDK are provided free of charge by Amazon - and accessed by the user directly for programming. The practice is usual: an AMI is free, the type of hardware reserved for the instance is not. Licenses are also included with AMIs, which is indeed a plentiful advantage.

In our case, it will be sufficient to start from the design already developed previously, registering an AFI (Amazon FPGA Image) and finally uploading it to the virtual machine. In this way, each AFI can be used several times and shared.

This section is a brief theoretical premise to understand the features of a majestic architecture, which cites many other usable services in addition to the one of FPGAs.

4.7.2 Test of the optimized code

The latest version of the code, the one reached in subsection 4.6.7, has therefore been synthesized for the target hardware platform. The process is performed by Vivado and takes a variable amount of hours (4-5 in this case), depending on any design parameters. From the reports relating to the execution on the actual FPGA, the execution time of a single kernel unit is a lot higher than expected, or at least

that estimated by Vivado HLS.

The execution time settles on $756.4ms$, only slightly more than 6.5× faster than the time of 5 of the CPU, and not even close to the result assumed by Vivado HLS of 50.3ms (15× slower).

The fact is certainly not encouraging since the best effort was put to achieve a pleasant performance, at least in terms of resource exploitation and parallelization (pipelining). The problem, however, derives from an aspect not predictable by Vivado HLS, i.e., the interfacing between host code and device. As mentioned previously, Vivado HLS generates the RTL relating to the kernel code regardless of what happens outside the device. Since that, it is unable to accurately estimate the problems related to the global memory (DRAM) access latency, the actual bottleneck of the design. HLS roughly estimates one reading per cycle to occur across the global bus, but not at whichever realistic clock frequency that happens.

The tracing reports summarize the read and write speeds between the kernel and global memory, together with the average transfer size. From here, it was realized that the significant problem, namely the access to individual locations of two arrays of symbols in global memory.

These arrays contain the simulation input symbols and are read-only. Their size is considerable, and the amount of local memory available is nowhere near enough to hold them in full. For the record, it was tried to copy them in their entirety inside URAM using a C kernel (arrays are mappable in URAM only with HLS attributes in the native C language, not with OpenCL attributes, as explained subsection 4.6.8). The result is a precise 200% occupation of memory, hence, impossible.

The convenience could come from the copy of array blocks for each antenna in reception and transmission, therefore 64 times in total in the design, but here comes a further problem.

Within the simulation, in fact, the symbols are read with addressing by columns, then with a regular jump between one data and the next. This address pattern does not allow data copying in the so-called burst mode, i.e., exploiting the full capacity of the bus by transferring blocks of data instead of individual data. Burst mode overcomes the problem of high access latency maximizing data throughput. For example, imagine that the AXI bus (configurable at 64, 128, 256, and 512 bits) has a bit width of 512 bits (default): it is possible to transfer 8 *double* data (64 bits each) thus reaching and exploiting the maximum capacity of the bus. The obstacle is that it is not possible at present to transfer the two arrays in blocks since these are accessed in non-consecutive locations.

Some steps were taken on the host side to overturn this situation, copying the arrays in global memory by columns and not by rows as initially done. The complexity of the pattern shifts on the host side, taking advantage of a much lower access latency (PCIe bus). This host adaption allowed access by rows on the device

side, slightly modifying the kernel code. Transfers between global and local memory now happen in bursts.

The HLS report for this version is reported for completeness, therefore including the added transfers. The theoretical performances worsen.

Latency (max)
17999865

Table 4.34: Latency (clock cycles)

With these new total latency, the estimated execution time grows to $17999865 \times 4ns = 71.99ms$.

	Latency		**Initiation Interval**		
Loop Name	Max	Iteration Latency	Achieved	Target	Trip Count
- rxAntLoop	17869758	8934879	-	-	2
+ txAntLoop	8873280	277290	-	-	32
++ copy	23	2	1	1	23
++ inSymb	15361	3	1	1	15360
++ coefLoop	95	74	1	1	23
++ symbolLoop	246137	380	2	1	122880
++ save	23	1	1	1	23
+ outSymb	61574	160	4	1	15360

Table 4.35: Latency of each loop (clock cycles)

Table 4.35 shows a slight increase in total cycles, due to the memory transfers added.

For the same reason, the use of BRAM also grows to over 50%.

Nonetheless, at the level of physical and real execution, it improves substantially. The execution time settles on an average of $132.2ms$, with an overall speedup of almost 38× compared to the best original software-only CPU execution, and more than 100× faster considering the run on the Amazon server's processor.

The other positive news derives from the fact that the report indicates, in addition to the total time of $132.2ms$, also the time of use of the compute unit, that is the pure computation unit. This time is $72.2ms$, and that approaches enormously the estimate by Vivado HLS. It can be assumed that the remaining $60ms$ are used to transfer the necessary data by interrupting the kernel execution. For example, think of the last modification made to the code: 64× in a kernel run the array of simulation input symbols (in BRAM) is replaced with new symbols. Also, output symbols are copied to global memory in two intervals, one in the middle of the

	BRAM_18K	**DSP48E**	**FF**	**LUT**	**URAM**
Total	2256	1026	270250	172119	0
Available	4320	6840	2364480	1182240	0
Utilization (%)	52	15	11	14	0

Table 4.36: Utilization estimates

simulation and one in the end.

This concludes the part closely related to the optimization work carried out. A summary of the results obtained and further considerations will be made during the final chapter of conclusions.

Chapter 5

Conclusions: results and future work

This final chapter summarizes the steps that are considered most important for obtaining the final version of the code. The results will then be compared with the starting data and expectations. The limitations that the new design introduces for the generality and customization of the simulation in question will also be highlighted. Finally will be indicated the remaining work to be done in order to complete and package the final version correctly for future use.

The working methodology was briefly discussed in the introduction chapter. It was mostly a succession of tests indicated by common practice. This practice is somehow suggested by the Xilinx Vivado HLS optimization manuals and put into practice in the various examples available to the user. These examples often relate to critical cases based on general/academic occurrences (e.g., FFT, DFT, multiplication between matrices). There is a profound gap between these cases and a purely real case like the one examined in the thesis. The code to be examined was indeed complex and varied. It includes short sections of logic and various floating-point operations with access to data structures through indirect addressing. A scenario that was not simple and immediate to manage.

The first improvements made are therefore derived from the use of the local block RAM on the FPGA and the partition of the arrays containing the data to facilitate the parallelization of tasks such as fast reads/writes and operations. The improvements have been made in a practical way, step by step, in order to become more familiar with the environment and the development tool faster.

The central part of the work was linked to the deepening of knowledge of the original code, with complete tracing about the data to be moved, such as read-only or read/write data and the study of the addressing mechanisms.

Aware of code problems and obstacles, we continued with a slight modification

of the algorithm in order to obtain the maximum achievable parallelization of the operations. That made it possible to arrive at an excellent first amount of total execution cycles, but limiting the simulation to some precise circumstances, in any case, modifiable by the user to return to a similar general case. In other words, it was necessary to structure the code manually based on a simulation variable (which became a constant), eliminating the possibility of parameterizing the code on it. If this parameter changes, someone should manually modify the code to follow this variation. This change certainly increases the complexity of the design by limiting its scalability. However, it has been a compromise accepted by the TIM supervisors in agreement with the original code designers.

The final part of optimization was suggested by Professor Lavagno, with whom the problems encountered were often discussed. The solutions implemented derived from a skillful look in more detail, often of log files or summary files at the register transfer level. That really helped to achieve satisfactory final results by making the latest changes, e.g., the use of redundant buffers.

Turning to an overview of the results, the initial expectations were extremely uncertain. On the one hand, there was low hope of achieving a substantial improvement, given the initial version of the code and the first results of the unfortunate hardware estimates. On the other hand, there were the indications of the vendor, which stated that for most designs the overall acceleration reaches one/two orders of magnitude compared to a new generation processing unit, therefore no more than 10 or 100 times.

The final version executed on the hardware settles on $132ms$. It has been said that this final time includes both the movement of the necessary data for the simulation and the pure algorithm execution.

During the discussion, two execution times were considered: the first is related to the execution of the original code on the CPU of a server at Politecnico di Torino, the other is the execution on an Amazon EC2 virtual machine, instance used for the test on a real FPGA. The most appropriate association would be to compare the post hardware acceleration result to this last execution time, the one on AWS, being the FPGA coupled with this processor on the machine. In this last case, the acceleration is excellent, being over $100\times$ (the average reference time on the CPU is 14.5 seconds).

On a general level, however, it would prove equally appropriate to compare with the lowest execution time obtained, i.e., the 5 seconds on the university machine. That allows us to follow the logic of the worst case, even if this reflection only holds halfway. There are no further details on the TIM data center servers, on which the simulation would run. The CPU could be more or less performing, or middle ground.

What can be said by comparing the two references is that to differentiate the execution times is the clock frequency of the two processors. The first is at 3.20GHz

and the simulation faster, while the second is at 2.30GHz. Another difference may be due to the Intel model (i7 vs. Xeon) and the fact that the first is leveraging DDR4 whether the second DDR3. These are general statements, and we do not want to go into the substance of the topic. In support of what has been said, especially on the ability to affect the pure clock frequency, is that the code is not concurrent, that is, it is executed as a single execution process (thread). In the worst considered case, i.e., where acceleration is minimum, this value is slightly higher than 35 times.

Note carefully that the acceleration value was carried out by measuring the execution time of the original code fragment only on the CPU, which then became kernel code. We are, therefore, only comparing the execution of the original function on the processor and the improved version performed on the hardware. No considerations were made regarding the execution of the improved code for the hardware on the CPU. Furthermore, the overhead introduced in the host code for the correct interface of the data between the host and the device was not considered.

In subsection 4.6.1, it was explained how changes to data structures were necessary to manage the transfer of data between CPU and FPGA. These changes are currently made before each kernel call and concern: split of single arrays of complex numbers into two arrays (real part and imaginary part), management of memory alignments. That introduces, at every kernel call, the execution of the code to make these adaptations.

Now the last steps and future work will be discussed in brief. These non-necessary loops can be removed by directly changing the data structures in the host C++ code. This will save the execution of cycles by directly modifying the source code, thus bypassing the need to adapt the arrays before each call to the kernel. However, this modification involves other changes to the code relating to the other functions of the channel model, and how the modified data structures are accessed.

A second change concerns the general packaging of the new and optimized code and other necessary minor changes. An example can be the addition of a function that deals with programming the device with the generated bitstream.

A third and final modification concerns the interfacing and coexistence between the kernel and that which Nasir Ali Shah has dealt with in his thesis [1]. The two kernels must coexist in the final design to obtain an overall appraisal of the total benefit of using the FPGA as a support for the calculation.

Bibliography

[1] N. A. Shah. «Optimization and Acceleration of 5G Link Layer Simulator». MS Thesis. Torino, Italy: Politecnico di Torino, Sept. 2019 (cit. on pp. 1, 22, 23, 67).

[2] M. Agiwal, A. Roy, and N. Saxena. «Next Generation 5G Wireless Networks: A Comprehensive Survey». In: *IEEE Communications Surveys Tutorials* 18.3 (thirdquarter 2016), pp. 1617–1655. ISSN: 2373-745X. DOI: 10.1109/COMST.2016.2532458 (cit. on pp. 4–6).

[3] The Mobile Network. «TMN Quarterly Magazine, Issue 18». In: May 2017. URL: https://the-mobile-network.com/2017/05/tmn-quarterly-magazine-issue-18/ (cit. on pp. 5, 8).

[4] EventHelix. «5G non-standalone access, Signaling flow for 5G access via LTE-5G NR dual connectivity (EN-DC)». In: *medium.com* (Mar. 2019). URL: https://medium.com/5g-nr/5g-non-standalone-access-4d48cea5db5f (cit. on p. 5).

[5] E. Hossain and M. Hasan. «5G cellular: key enabling technologies and research challenges». In: *IEEE Instrumentation Measurement Magazine* 18.3 (June 2015), pp. 11–21. ISSN: 1941-0123. DOI: 10.1109/MIM.2015.7108393 (cit. on pp. 5, 6, 8).

[6] F. Boccardi, R. W. Heath, A. Lozano, T. L. Marzetta, and P. Popovski. «Five disruptive technology directions for 5G». In: *IEEE Communications Magazine* 52.2 (Feb. 2014), pp. 74–80. ISSN: 1558-1896. DOI: 10.1109/MCOM.2014.6736746 (cit. on pp. 6, 7).

[7] A. Gupta and R. K. Jha. «A Survey of 5G Network: Architecture and Emerging Technologies». In: *IEEE Access* 3 (2015), pp. 1206–1232. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2015.2461602 (cit. on p. 6).

[8] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. K. Soong, and J. C. Zhang. «What Will 5G Be?» In: *IEEE Journal on Selected Areas in Communications* 32.6 (June 2014), pp. 1065–1082. ISSN: 1558-0008. DOI: 10.1109/JSAC.2014.2328098 (cit. on p. 7).

[9] Heidi Vella. *5G vs 4G: what is the difference?* URL: https://www.raconteur.net/technology/4g-vs-5g-mobile-technology (cit. on p. 7).

[10] S. Sun, G. R. MacCartney, and T. S. Rappaport. «A novel millimeter-wave channel simulator and applications for 5G wireless communications». In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–7. DOI: 10.1109/ICC.2017.7996792 (cit. on p. 8).

[11] Nafiz Imtiaz Bin Hamid, Mohammad Kawser, and Md. Ashraful Hoque. «Coverage and Capacity Analysis of LTE Radio Network Planning considering Dhaka City». In: *International Journal of Computer Applications* 46 (May 2012), pp. 49–56. DOI: 10.5120/6989-9604 (cit. on p. 8).

[12] R. Jain. «Channel Models A Tutorial». In: (Feb. 2007). URL: https://www.cse.wustl.edu/~jain/cse574-08/ftp/channel_model_tutorial.pdf (cit. on p. 9).

[13] Pekka Kyösti. «Radio channel modelling for 5G telecommunication system evaluation and over the air testing.» Academic dissertation. University of Oulu, P.O. Box 8000, FI-90014 University of Oulu, Finland: University of Oulu Graduate School, May 2018 (cit. on pp. 9–11).

[14] P. Ferrand, M. Amara, S. Valentin, and M. Guillaud. «Trends and challenges in wireless channel modeling for evolving radio access». In: *IEEE Communications Magazine* 54.7 (July 2016), pp. 93–99. ISSN: 1558-1896. DOI: 10.1109/MCOM.2016.7509384 (cit. on p. 9).

[15] 3GPP. *Technical Specification Group Radio Access Network; Study on channel model for frequencies from 0.5 to 100 GHz (Release 14)*. Technical Report (TR) 38.901. Version 14.3.0. 3rd Generation Partnership Project (3GPP), Dec. 2017 (cit. on pp. 10–12, 25).

[16] Y. Wang, J. Xu, and L. Jiang. «Challenges of System-Level Simulations and Performance Evaluation for 5G Wireless Networks». In: *IEEE Access* 2 (2014), pp. 1553–1561. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2383833 (cit. on p. 10).

[17] Yawei Yu, Jianhua Zhang, Mansoor Shafi, Min Zhang, and Jawad Mirza. «Statistical Characteristics of Measured 3-Dimensional MIMO Channel for Outdoor-to-Indoor Scenario in China and New Zealand». In: *Chinese Journal of Engineering* 2016 (Mar. 2016), pp. 1–10. DOI: 10.1155/2016/1317489 (cit. on p. 11).

[18] G. Borriello, C. Ebeling, S. A. Hauck, and S. Burns. «The Triptych FPGA architecture». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.4 (Dec. 1995), pp. 491–501. ISSN: 1557-9999. DOI: 10.1109/92.475968 (cit. on p. 13).

[19] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges*. now, 2008. ISBN: null. URL: https://ieeexplore.ieee.org/document/8187326 (cit. on p. 14).

[20] Chris Jackson. «Investigating Serial Microprocessors for FPGAs». In: (Mar. 2020) (cit. on p. 15).

[21] Alexander Marquardt, Vaughn Betz, and Julie Rose. «Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density». In: Jan. 1999. DOI: 10.1145/296399.296426 (cit. on p. 15).

[22] Steve Kilts. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007 (cit. on p. 15).

[23] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. «An Introduction to High-Level Synthesis». In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 8–17. ISSN: 1558-1918. DOI: 10.1109/MDT.2009.69 (cit. on pp. 16, 17).

[24] Michael McFarland and Alice Parker. «Tutorial on High-Level Synthesis.» In: Jan. 1988, pp. 330–336 (cit. on p. 16).

[25] G. Martin and G. Smith. «High-Level Synthesis: Past, Present, and Future». In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 18–25. ISSN: 1558-1918. DOI: 10.1109/MDT.2009.83 (cit. on p. 17).

[26] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. «LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems». In: *ACM Transactions on Embedded Computing Systems (TECS)* 13 (Sept. 2013). DOI: 10.1145/2514740 (cit. on p. 17).

[27] Dominik Grewe and Michael O'Boyle. «A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL». In: Mar. 2011, pp. 286–305. DOI: 10.1007/978-3-642-19861-8_16 (cit. on p. 18).

[28] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. «Synthesis of Platform Architectures from OpenCL Programs». In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. May 2011, pp. 186–193. DOI: 10.1109/FCCM.2011.19 (cit. on pp. 18, 19).

[29] Xilinx. *SDAccel Development Environment Help (v2018.2)*. Version v2018.2. Aug. 2018. URL: https://www.xilinx.com/html_docs/xilinx2018_2/sdaccel_doc/index.html (cit. on pp. 19, 56).

[30] *OpenCL*. URL: https://de.wikipedia.org/wiki/OpenCL (cit. on p. 20).

[31] *BittWare XUPSV2 | Ultra high-speed network interface | Xilinx UltraScale+ FPGA | 2x 100GbE | 460 GBps memory bandwidth | 16 GB | Low-profile PCIe*. URL: https://www.zerif.co.uk/legacy-and-exotic/bittware-xupsv2-xilinx-virtex-2x-qsfp-16-gb (cit. on p. 27).

[32] Xilinx. *UltraScale Architecture Memory Resources, User Guide*. Version v1.10. Feb. 2019. URL: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf (cit. on p. 28).

[33] *Xilinx Announces Virtex UltraScale+, the World's Largest FPGA*. URL: https://www.techpowerup.com/258678/xilinx-announces-virtex-ultrascale-the-worlds-largest-fpga (cit. on p. 29).

[34] Xilinx. *Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit*. URL: https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html (cit. on p. 29).

[35] Xilinx. *Large FPGA Methodology Guide, Including Stacked Silicon Interconnect (SSI) Technology*. Version v14.3. Oct. 2012. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf (cit. on p. 28).

[36] Xilinx. *Vivado Design Suite User Guide, High-Level Synthesis*. Version v2018.2. July 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf (cit. on pp. 31, 51).

[37] Xilinx. *SDx Pragma Reference Guide*. Version v2018.2. July 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1253-sdx-pragma-reference.pdf (cit. on p. 34).

[38] Khronos Group. *OpenCL 1.0 Reference Pages*. URL: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/ (cit. on p. 56).