Investigating scaling techniques and the cost-efficiency of distributed to single FPGA compositions for Full Waveform Inversion

by

Luc Dierick

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 7, 2022 at 10:00 AM.

Student number:	4445120	
Project duration:	November 8, 2021 – July 7, 2022	
Thesis committee:	Prof. Dr. Z. Al-Ars,	TU Delft, supervisor
	Prof. Dr. M. A. Zuñiga Zamalloa,	TU Delft
	Ir. J. Petri-König,	AMD
	Ir. G. Gunay,	ALTEN

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

During my studies, my interests shifted from high-level software engineering to embedded systems. This eventually led me to fulfill my thesis with the Accelerated Big Data Systems (ABS) group of the Computer Engineering (CE) department. I stepped out of my comfort zone by taking on a hardware-oriented project. Exploring the complexities of hardware design sometimes left me clueless, but in the end proved to be a very rewarding journey. Fortunately, I was not facing the endeavors of this project alone. I would like to thank my parents, girlfriend, brother, sister, and friends who have supported my decisions and provided me with motivation and opportunities to decompress during these sometimes stressful times. Throughout this project, my supervisors provided me with unwavering support, mentoring, and motivation. For that, I express my sincere gratitude to Zaid Al-Ars, Jakoba Petri-König, and Gokhan Gunay. A special thank you is to Jakoba as she continued to supervise my work, respond to my countless messages, and provide me with excellent feedback even after leaving the TU Delft and starting a new job. Lastly, I would like to thank Joost Hoozemans and André Prins for their involvement in the choice of topic for this project and ALTEN for providing me with a place to work, fun colleagues and an introduction into the technical consulting industry.

Luc Dierick
Rotterdam, June 2022

Abstract

In recent years, the big data era has produced an increasing volume and complexity of data that requires processing. To analyze and process these large amounts of data, applications are being scaled on large clusters using distributed data processing frameworks. A more recent trend utilizes hardware accelerators to offload computationally intensive tasks and reduce compute time and energy consumption. As a result, a rapid growth of data center deployment containing heterogeneous compute infrastructures is observed. Alternative to the more commonly used general-purpose GPUs (GPGPUS), the field programmable gate array (FPGA) is becoming an increasingly popular choice of accelerator. Its effectiveness to accelerate highly parallel applications in combination with the flexibility due to its reconfigurable nature make it well suited for a wide range of applications. As a spatial compute resource, the problem size a single FPGA can process is bounded by the available programmable logic and memory. However, applications that do not require the full resources of an FPGA can be vertically scaled by instantiating multiple instances of the hardware design on a single node. A barrier in the adoption of FPGAs is formed by the complexity of hardware design which requires in depth hardware-specific expertise. Additionally, integrating FPGAs in distributed data processing frameworks is a challenge on itself.

These challenges are being addressed in two directions. High level synthesis (HLS) tools and compilers are being developed to decrease the complexity of hardware design by allowing users to develop FPGA designs in high level languages. Additionally, there is an increased availability of ready-to-use FPGA designs for common applications in hardware libraries such as Vitis libraries [1].

To aid the adoption of FPGAs and improve their accessibility, this work presents OctoRay: a python framework with a focus on ease-of-use that allows users to flexibly and transparently scale applications both vertically and horizontally on FPGA clusters. Scaling a binarized convolutional neural network (CNN) with OctoRay resulted in performance improvements linear to the number of nodes, or copied instances applied. The framework was also used to analyze the cost-efficiency of a cluster of low-end PYNQ-Z1 FPGAs compared to a data center class Alveo U280 FPGA. A partly in hardware accelerated implementation of Full Waveform Inversion (FWI), a seismic imaging algorithm, was developed and used to conduct the investigation. It was concluded that 32 PYNQ-Z1s are required to match the performance of a single Alveo U280 FPGA. An important bottleneck in the performance of the PYNQ-Z1s was the low-performance host processor on which a significant portion of FWI was executed. The small number of resources available on a PYNQ-Z1 limited the attainable accuracy of FWI to a bare minimum. The FWI hardware design with the same specifications made for the high-end FPGA only utilized a fraction of its resources, far from harnessing its full potential. It was concluded that, unlike FWI, applications that do not require the abundance of resources a high-end FPGA offers, but do benefit from rapid development cycles and low energy consumption are suited for a distributed low-end FPGA composition.

List of Acronyms

ADAS Advanced Driver-Assistance Systems.

ASIC Application Specific Integrated Circuit.

CLB Configurable Logic Block.

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

DNN Deep Neural Network.

FPGA Field Programmable Gate Array.

FWI Full Waveform Inversion.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High Level Synthesis.

IC Integrated Circuit.

IP Intellectual Property.

RTL Register Transfer Level.

SoC System on Chip.

VHDL Very high speed integrated circuit Hardware Description Language.

List of Figures

2.1 Simplified visualization of the components of an FPGA, figure altered from [15] 6
2.2 FPGA design flow phases 7
2.3 Dependency graph of parallelism example application. 9
2.4 High level schematic of task parallelism. 10
2.5 High level schematic of data parallelism. 11

3.1 A high-level overview of the FWI algorithm. 14
3.2 The synthetically generated acoustic model with a homogeneous background of $c_0 = 2000m/s$, embedded in the background are four layers with constant velocities from top to bottom of respectively 2200, 2233, 2300 and 2266 m/s. The differences are displayed by means of the contrast function and represent chi values. The grid is scaled where 1 grid length in the figure represents 5 meters. 14
3.3 A high-level overview of the FWI algorithm. 15
3.4 Visualization of the outline of a CNN, altered from [36] 17

4.1 A high-level representation of the stack necessary to scale applications on FPGAs. . . . 20
4.2 An overview of the Xilinx Runtime Library (XRT) adapted from [40] 22
4.3 The different classes of Dask modules and how they relate, figure adapted from [42] . . 24
4.4 A visualization of the solution architectures for FWI on Alveo U280 and PYNQ-Z1 and CNN on Alveo U280. 26

5.1 An overview of the percentage of run time occupied by the forward, cost, and combined functions and the iterative loop for grid sizes 500, 1000, and 10000 and resolutions (sources*receivers*frequencies), $100 = (5 * 5 * 4)$, $1000 = (10 * 10 * 10)$, $6000 = (15 * 15 * 20)$ and $10000 = (20 * 20 * 25)$. 28
5.2 The Zynq Processing System IP, taken from [44] 30
5.3 Visualization of block and cyclic array partitioning of an 8-element array with factor 2, taken from [46] 32
5.4 A visualization of the effect of pipelining, taken from [48]. 33
5.5 The general architecture of the OctoRay framework, figure ada[ted from [53] 39

6.1 Topology of hardware platforms and applications. 44
6.2 An overview of the power measurement setup for PYNQ-Z1 from the DTG group of University of Kassel. 45
6.3 The real subsurface grid of the Delphi temple (left), the reconstructed grid (center), and the difference between the real and reconstructed grid (right) are expressed in chi. The C++ FWI implementation was applied on the full data set of grid size $64 * 32 = 2048$ with a resolution of sources, receivers, and frequencies $20 * 20 * 15 = 6000$. 48
6.4 The real subsurface grid of the Delphi temple (left), the reconstructed grid (center), and the difference between the real and reconstructed grid (right) are expressed in chi. The C++ FWI implementation was applied on the full data set of grid size $64 * 32 = 2048$ with a resolution of sources, receivers, and frequencies $7 * 7 * 6 = 294$. 48
6.5 The real subsurface grid of the Delphi temple (left), the reconstructed grid of four independently inverted layers merged back together (center) and the difference between the real and reconstructed grid (right) is expressed in chi. The C++ FWI implementation was applied four times on $64 * 8 = 512$ grid size layers of the total data set. Each layer was inverted with a resolution of sources, receivers and frequencies $7 * 7 * 6 = 294$ and are separated by white lines in the figure. 49

6.6 The accuracy of the CPU implementations is expressed through the contrast function chi (left). The execution time of the C++ implementation for the independent and summed layers of the merged model with a resolution (R) of 294 and of the complete data set with resolutions R=294 and R=6000 (right). 49
6.7 An overview of the execution time spent in the hardware-accelerated functions for the PYNQ-Z1 and Alveo U280 FPGAs and CPU python implementations. The high utilization PYNQ-Z1 hardware design from Table 6.1 with resolution 294 was used. The python FWI implementation was used for both CPU and PYNQ-Z1. These results do not take initialization and downloading the bitstreams to the FPGAs into account. 51
6.8 The left plot shows the performance of FWI with a resolution of 300 and grid size 100 when scaled on 1, 2, and 4 PYNQ-Z1s. The right plot shows the performance of the same FWI configuration for the python and C++ CPU implementation, the Alveo U280, and 32 PYNQ-Z1s. 51
6.9 A comparison of vertical, horizontal, and no scaling for 1000 and 5000 resolution FWI configurations on Alveo U280. R denotes the resolution, G the batch of grid cells per compute unit or per node, and CU the number of compute units. 53
6.10 The performance of an 8000 resolution and 100 grid size FWI configuration. The comparison is made for the C++ and python implementation on CPU, the former in combination with OpenCL and the latter in combination with OctoRay and subsequently pynq, on an Alveo U280 FPGA. 53
6.11 The overhead in milliseconds induced by Dask for 100, 1000 and 5000 total subsurface grid sizes executed by an 8000 resolution 100 grid size FWI configuration. 54
6.12 The total design-build duration is divided into kernel synthesis and the remaining build duration for three hardware designs of FWI. 56
6.13 The global trends in user internet traffic, data center workload, and energy consumption. The numbers in this graph are relative to 2010. Data was taken from IEA [58] 57
6.14 The power and energy consumption of the PYNQ-Z1, and Alveo U280 FPGAs. The power is measured for total grid sizes 100, 1000, and 5000, decomposed in batches of 100 grid cells. The energy is calculated by using the time spent in the accelerated part of the algorithm as the power is also only measured over that part. 58
6.15 The throughput in images per second of 1 dask worker executing the CNN application with a 1 instance hardware design for batch sizes 1, 100, 500, 1000, and 10000 of 32x32 RGB images. The dask scheduler and client were on the HACC alveo3b with IP 10.1.212.126 and the dask worker on alveo3c with IP 10.1.212.127. 59
6.16 The total execution time and throughput for CNN designs with 1,2,4,8 and 10 CUs on 1 node with 1 dask worker and a CNN design with 1 CU on 4 nodes and dask workers. The numbers represent the average of 5 runs for batch sizes 1000, 5000, 10000, 50000, and 100000 of 32x32 RGB images from the cifar-10 dataset. 59
6.17 The design-build duration for 1, 2, 4, 8, and 10 CU CNN designs decomposed into the different components of the total design-build. 60
6.18 The throughput in images per second of 1 dask worker executing the CNN application with a 1 instance hardware design for batch sizes 1, 100, 500, 1000 and 10000 of 32x32 RGB images. 61

A.1 A diagram of the ALTEN provided implementation of FWI using the finite difference forward model and a conjugate gradient descent inversion model. 68

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Challenges . . . 2
  - 1.2.1 Accessibility . . . 2
  - 1.2.2 Cost-efficiency . . . 2
- 1.3 Problem statement & research questions . . . 3
- 1.4 Solutions and contributions . . . 3
- 1.5 Thesis outline . . . 4

2 Background **5**
- 2.1 Field programmable gate arrays . . . 5
  - 2.1.1 Design flow . . . 6
  - 2.1.2 SoC FPGAs . . . 7
  - 2.1.3 Platform resources . . . 8
  - 2.1.4 Direct memory access . . . 8
- 2.2 Frameworks and toolchains . . . 8
  - 2.2.1 Vivado design suite . . . 9
  - 2.2.2 Vitis unified software platform . . . 9
- 2.3 Scalability . . . 9
  - 2.3.1 Task parallelism . . . 9
  - 2.3.2 Data parallelism . . . 10
- 2.4 Related work . . . 11

3 Application domains **13**
- 3.1 Full Waveform Inversion . . . 13
  - 3.1.1 Background & implementation . . . 13
  - 3.1.2 Scalability characteristics . . . 16
- 3.2 Convolutional Neural Network . . . 17
  - 3.2.1 Background & implementation . . . 17
  - 3.2.2 Scalability characteristics . . . 18

4 Alternative solutions **19**
- 4.1 Hardware-design . . . 20
  - 4.1.1 Custom hardware design . . . 20
  - 4.1.2 Hardware designs from libraries . . . 21
  - 4.1.3 Compiler-generated hardware designs . . . 21
  - 4.1.4 Evaluation . . . 21
- 4.2 Hardware communication . . . 22
  - 4.2.1 XRT . . . 22
  - 4.2.2 OpenCL . . . 23
  - 4.2.3 PYNQ . . . 23
  - 4.2.4 Evaluation . . . 23
- 4.3 Parallelization framework . . . 23
  - 4.3.1 Apache Spark . . . 24
  - 4.3.2 Dask . . . 24
  - 4.3.3 Evaluation . . . 24
- 4.4 Interface . . . 25
  - 4.4.1 Programming languages . . . 25
  - 4.4.2 Evaluation . . . 25

4.5 Conclusion . . . 26
4.5.1 FWI on PYNQ-Z1 . . . 26
4.5.2 FWI on Alveo U280 . . . 26
4.5.3 CNN on Alveo U280 . . . 26

5 Solution architectures and implementations **27**
5.1 FWI hardware design . . . 27
5.1.1 Algorithm analysis . . . 27
5.1.2 Hardware kernels . . . 28
5.1.3 Design implementation . . . 29
5.1.4 Design optimization . . . 31
5.2 Hardware communication . . . 34
5.2.1 PYNQ . . . 34
5.2.2 OpenCL . . . 36
5.2.3 Multiple compute units . . . 37
5.3 Interface . . . 38
5.4 OctoRay . . . 38
5.4.1 Deployment . . . 40
5.4.2 Flexibility and usability . . . 41
5.4.3 Accessibility . . . 43

6 Results **44**
6.1 Experimental setups for FWI . . . 44
6.1.1 CPU . . . 44
6.1.2 PYNQ-Z1 OctoRay . . . 45
6.1.3 PYNQ-Z1 power measurement . . . 45
6.1.4 Alveo U280 . . . 46
6.2 Experimental setup CNN . . . 47
6.3 FWI results . . . 47
6.3.1 Domain decomposition . . . 47
6.3.2 Scalability . . . 50
6.3.3 Performance . . . 53
6.3.4 Cost and utilization . . . 54
6.3.5 Design-build duration . . . 56
6.3.6 Power . . . 56
6.4 CNN results . . . 58
6.4.1 Scalability . . . 59
6.4.2 Build design time . . . 60

7 Conclusions and future work **62**
7.1 Conclusions . . . 62
7.1.1 Accessibility . . . 62
7.1.2 FWI acceleration . . . 63
7.1.3 Cost-efficient compositions . . . 63
7.1.4 Scalability . . . 64
7.2 Future work . . . 64

A Full Waveform Inversion **66**
A.1 Cost Function C++ . . . 66
A.2 Forward Function C++ . . . 67
A.3 Implementation diagram . . . 68

1 Introduction

1.1. Context

With the growing deployment of data centers and heterogeneous compute infrastructures, the Field Programmable Gate Array (FPGA) is becoming a serious contributor to the Big Data processing paradigm. This has been emphasized in recent years as leading chip-makers Intel and AMD have, respectively, acquired FPGA vendors Altera and Xilinx. Cloud computing giants such as Amazon AWS, Microsoft Azure, and Google Cloud have adopted FPGAs into their infrastructures right next to silicon alternatives, CPUs, and GPUs. Microsoft, for example, exploits FPGAs for its low-latency performance in Deep Neural Network evaluations and even Bing's search ranking [2].

FPGAs have shown in the past that they are a promising alternative to further accelerate applications, especially highly parallelizable ones. For example, in research areas such as genomics, FPGAs are used to accelerate key algorithms that need to process increasingly larger data sets [3]. Also in distributed Big Data applications FPGAs have shown their worth, for instance, to accelerate compression and decompression algorithms that reduce data storage space and data transmission bandwidth [4]. Unfortunately, due to the high complexity of FPGA development, exploitation of FPGAs used to be reserved for the relatively small group of engineers with in-depth hardware knowledge. Companies such as AMD and Intel have been releasing software tools and frameworks to enable developers with little to no hardware knowledge to use FPGAs. Additionally, in the academic world efforts are being made to further integrate FPGAs into heterogeneous compute infrastructures. Hoozemans et al. provide an overview of the current opportunities, challenges, and their potential solutions for Big Data Analytics acceleration with FPGAs in [5].

Despite these initiatives, there is still a lot to be done to lower the entry barriers and improve accessibility. Compared to CPUs and GPUs, the FPGA ecosystem is still very much in its infancy. Frameworks such as InAccel [6], are proprietary and designed to support expensive data center-class FPGAs. It should be noted that historically, the electrical engineering (EE) community, contrary to the software (SW) community, has been averse to the concept of open-source work [7]. To aid combat this aversity and grow the FPGA ecosystem in a healthy way, there is a need for more open-source frameworks to help a broader public scale and accelerate applications on FPGAs.

In 2020 the largest share of the FPGA market was held by low-end accelerators [8]. This trend is predicted to continue due to different benefits such as low cost, higher energy efficiency, and reduced complexity when using low-end FPGAs compared to their data center-class counterparts. Some application classes, used in sectors like the automotive business and Internet of Things require low-end FPGAs for the previously mentioned benefits. Other classes require the computing performance high-end FPGAs have to offer.

A common aspect, the entire range from low to high-end FPGAs have, is their potential to accelerate highly parallel applications. The size and parallelism available in modern-day Big Data applications

often exceed the computing power of a single FPGA node. By scaling these applications on multiple FPGA nodes, the parallelism can be further exploited resulting in potentially even more significant benefits. This work was performed in collaboration with ALTEN Netherlands and the Accelerated Big Data Systems (ABS) group from TU Delft. ALTEN provided a computationally demanding application with highly parallel features called Full Waveform Inversion to conduct the investigations in this thesis.

1.2. Challenges

Two of the main challenges in scaling data analytics applications on FPGAs are accessibility and cost-efficiency.

1.2.1. Accessibility

The FPGA market has known its challenges since the invention of the FPGA in the early 1980s. In the first 2 decades of development, customer demand dictated the need for FPGAs with more resources to fit their designs. Vendors reacted to this demand by developing modern-day data center-class FPGAs that offer sufficient capacity for the majority of applications [9].

Due to these technological developments, the challenges have shifted from capacity to accessibility and large-scale integration in the data processing paradigm. This challenge is two-fold, due to the complexity and steep learning curve of FPGA design, it is reserved for experienced hardware designers. In addition, there is an absence of support in the form of open-source tools and frameworks for the adoption of application scaling on FPGAs. Developers and engineers with little to no hardware knowledge should be able to transparently and seamlessly scale applications by exploiting the capabilities of FPGAs.

There are current initiatives to increase the accessibility of FPGAs through the introduction of tools such as High Level Synthesis (HLS). HLS enables developers to synthesize FPGA designs from C-like code. Although a step in the right direction, designs generated with HLS are non-optimal implementations and still require in-depth hardware knowledge to fine-tune them into efficient FPGA designs. Another initiative is the concept of build-once, re-use often, where efficient FPGA designs for common applications such as Neural Networks are developed by experienced hardware designers to be made available for the general public. While this concept offers a lot of potential in FPGA adoption, there remains a need for frameworks and support in the data processing toolchains to exploit these ready-to-use hardware designs in a scalable way.

1.2.2. Cost-efficiency

Modern-day data center-class accelerator cards like the Alveo family offer a lot of resources that, if used correctly, can attain enormous performance, throughput, and latency boosts compared to CPUs [10]. At first sight these high-end accelerators may seem adequate for most application classes, but there is no free lunch in engineering and the benefits come at costs such as higher power consumption, costly manufacturing, and expensive development cycles.

Costly manufacturing

Manufacturing very large integrated circuits is inherently more costly than smaller integrated circuits due to the larger amount of silicon, wafer, and other materials needed. Furthermore, there is a higher chance of defects and yield problems in larger circuits.

Expensive development cycle

Not only does the cost of manufacturing increase when working with large FPGAs, but other stages of the development cycle also become more expensive. The FPGA design flow, more elaborately discussed in Chapter 2, consists of 4 main phases: design entry, synthesis, implementation, and, programming. In the first phase, a hardware design is written in a Hardware Description Language. In the synthesis phase, this design is translated to a netlist, a collection of logic elements, interconnection resources, and other components necessary to build the design. The implementation phase places and routes the elements from the netlist on the FPGA resources while taking timing, location, and other

constraints into consideration.

The synthesis and implementation phases are optimization problems that increase drastically in complexity as the size of the FPGA and design grows. This results in considerably longer, memory and computationally demanding development cycles, often orders of magnitude larger than low-end FPGAs experience. While the development cycle of a low-end FPGA can be performed on an average personal computer, data center-class FPGAs require server-grade machines.

Power consumption
Even though FPGAs are considered a silicon alternative that consumes little energy, especially compared to GPUs, there is still a significant difference between small and large FPGAs. Larger FPGAs such as an Alveo U280 have a maximum total power of 225W [11]. A smaller data center-class FPGA, the Alveo U50 has a maximum total power of 75W, which is significantly less, but still an order of magnitude larger than the 10W maximum total power of a low-end accelerator like the PYNQ-Z1 [12].

1.3. Problem statement & research questions

In this work, two challenges in the adoption of FPGAs in the data processing paradigm are investigated. There is a lack of tools, frameworks and support for users to easily accelerate and scale data analytics applications on FPGAs. By solving this challenge and enabling seamless scalability over multiple FPGA nodes it becomes possible to consider an alternative cost-efficient composition of multiple low-end FPGAs instead of a single high-end FPGA. This project aims to provide recommendations and solutions to these challenges by using FWI as the primary use case. More precisely, the following research question are answered in this thesis:

1. How can FPGAs be used to scale data analytics applications in a transparent and accessible way?
2. Can FWI be accelerated and scaled on FPGAs?
3. Is a composition of low-end FPGAs a viable alternative to a data center-class FPGA in terms of performance, power efficiency, and cost when accelerating FWI?
4. How does vertical and horizontal scaling affect the performance of FPGA accelerated applications?

1.4. Solutions and contributions

To answer the above-stated research questions several solutions were implemented. The challenge of FPGA accessibility and adoption was addressed by developing OctoRay, a python-based framework that enables users to transparently scale applications on FPGA clusters. A hardware design for the FWI use case was developed and used in combination with OctoRay to accelerate and scale the application on multiple FPGAs. Subsequently it was possible to investigate the viability of using multiple low-end FPGAs alternative to a high-end FPGA for FWI acceleration. The effect of horizontal and vertical scaling was investigated for FWI. An application with different scalability characteristics, a Convolutional Neural Network (CNN) for image classification, was introduced to extend the investigation.

In summary, the contributions of this work are:

1. OctoRay, a flexible framework that enables users to scale applications horizontally and vertically with a focus on ease-of-use.
2. Design, implementation and evaluation of the FWI algorithm for low-end PYNQ-Z1 and high-end Alveo U280 FPGAs.
3. An analysis of the effect of horizontal and vertical scaling for applications with different scalability characteristics CNN and FWI.
4. An evaluation of the viability of a multiple low-end FPGA composition as alternative to a high-end FPGA for FWI acceleration.

1.5. Thesis outline

In Chapter 2, the concepts, tools, and complementary subjects that serve as a foundation for the understanding of the choices, solutions, and implementations presented in this work are introduced. In Chapter 3, the two applications that were used to benchmark and validate the solutions are presented. This chapter also discusses the characteristics of these applications in terms of scalability. Chapter 4 presents the solutions space considered, successively the chosen solution architectures and implementations are explained in Chapter 5. The experiment setups and the gathered results are presented and discussed in Chapter 6. In Chapter 7, the final chapter of this work, the conclusions made are discussed and recommendations for future work are proposed.

2

Background

This chapter explains the concepts, tools, and other subjects that were used to analyze the problems stated in Chapter 1 and to build the proposed solutions. First, Section 2.1 introduces the Field Programmable Gate Array followed by a description of the design flow and the resources for the targeted platforms in this work. Secondly, the tools from AMD used to implement the solution architectures are discussed in Section 2.2. Having established the platforms and tools required for a single node implementation, concepts to apply scalability are introduced in Section 2.3. Lastly, work related to the adoption of FPGAs with similar approaches as applied in this work are discussed in Section 2.4

2.1. Field programmable gate arrays

Field Programmable Gate Arrays were invented in the early 1980s with the goal to accelerate the design process of Application Specific Integrated Circuits (ASIC). The use of FPGAs enabled engineers to design, program, validate and verify hardware designs without having to manufacture actual Integrated Circuits.

As FPGA development matured over the years, they appeared to be useful for much more than simply accelerating ASIC design. In modern days, there are use cases in many different industries for FPGAs. Data centers nowadays are made up of heterogeneous computing infrastructures that aim to make optimal use of the different available hardware options. Like GPUs and ASICs, FPGAs complement CPU clusters in this infrastructure by processing computationally demanding workloads. FPGAs differentiate themselves from ASICs in applications that require rapid development cycles and from GPUs in applications that require low latency as well as energy efficiency [13]. In the automotive industry, FPGAs have taken the upper hand over ASICs when developing Advanced Driver-Assistance Systems, especially due to the reconfigurability characteristic that allows ADAS engineers to easily build on previous designs and reduce the time to market tremendously [14]. As the focus of this work is not on the optimization of FPGA design, only a high-level overview of the internals of an FPGA are presented, the design flow approach, and finally, we introduce the two specific FPGAs that were used during experiments.

A Field Programmable Gate Array is a unique type of IC that is made up of Configurable Logic Blocks (CLB) and I/O cells which are all connected in a matrix form through interconnection resources, visualized in Figure 2.1. To create functionality, the previously mentioned components are combined into blocks of logic and data. In software design, functions and classes are used to increase code reusability. In hardware design the same applies, to increase the reusability of designs the functional blocks of logic and data are bundled into so-called Intellectual Property (IP) cores. These cores should, when correctly designed, offer portability in a vendor-agnostic manner so that they can be easily integrated in any design methodology. Besides design reusability, IP cores are also fundamental in enabling users with little to no hardware design experience to utilize FPGA functionality.

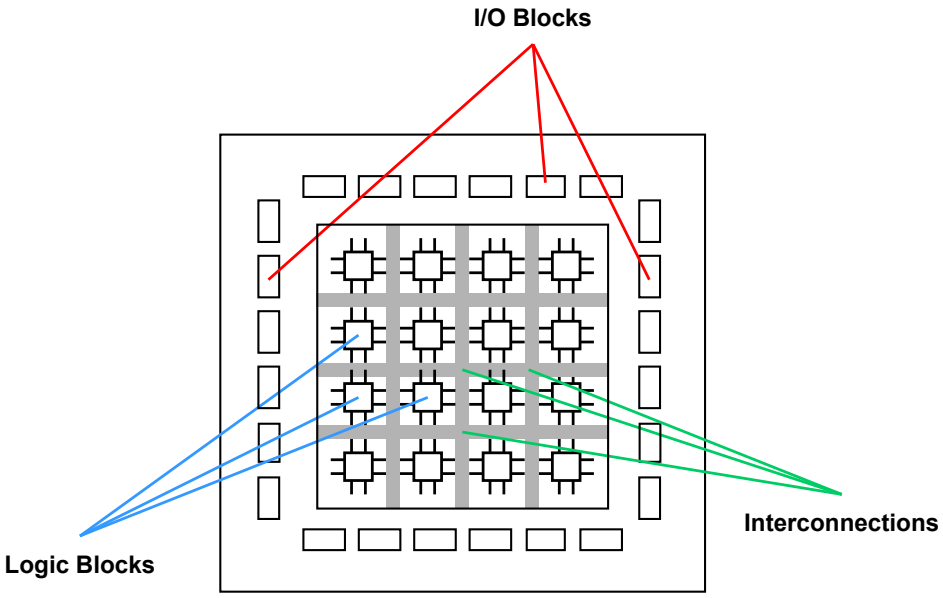

Figure 2.1: Simplified visualization of the components of an FPGA, figure altered from [15]

2.1.1. Design flow

Different hardware platforms adhere to different design flows to utilize them. Programming CPUs is slightly different from GPUs, but both design flows have in common that they result in a set of software instructions that are executed by the designated hardware platform. The design flows for ASICs and FPGAs are different in that sense, because they result in an IC. The difference between ASICs and FPGAs, is that the former draws the resulting circuit directly into silicon while the latter creates the circuit by connecting the CLBs and other components.

The main phases in FPGA design flow are shown in Figure 2.2. The first phase, design entry, consists of defining the requirements of the system. This can be done with schematics, where the design is drawn with gates and wire, or the more common approach, by describing the design with a Hardware Description Language. Specifically and unambiguously describing the behavior of an IC is not an easy task. Historically, people started describing them on a component level and later rose to a logical level. In the 1980's HDLs such as VHDL and Verilog were introduced. These languages allowed engineers to explicitly describe the behavior of large IC's and the opportunity to simulate and test these systems. The latter significantly reduces the cost of development because circuits could now be validated and tested on performance before being physically created.

Quickly after the introduction of HDLs, engineers started working on tools to abstract even further away from low-level hardware design. They introduced High Level Synthesis which takes a high-level specification of the hardware design in a C-like programming language and compiles it into a Register Transfer Level written in a HDL. High Level Synthesis is still a hot topic that is constantly being developed with the goal to enable software engineers to be able to accelerate their applications on FPGAs at the same rate as experienced hardware designers would. [16]

The next phase in FPGA design flow is synthesis, where the designed system is translated from schematics or HDL into a netlist. A netlist is the collection of logic elements, interconnection resources and other components necessary to build your design.

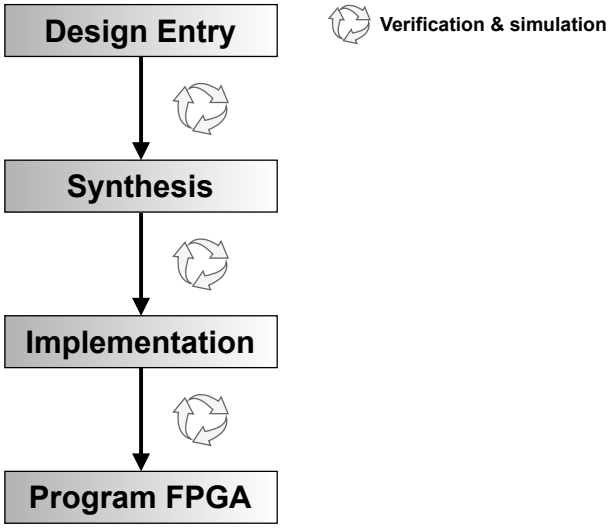

Figure 2.2: FPGA design flow phases

In the following phase, the components from the netlist are placed and routed onto the matrix of FPGA resources. In this step, the constraints a designer may have posed such as timing, or specific placements are taken into account. Both the Implementation and Synthesis phases apply optimization techniques to use the resources as efficiently as possible. The time, memory, and computing resources necessary to complete these phases increase with the complexity of the design and constraints, as well as with the total amount of FPGA resources available. With complex designs and data center-class FPGAs, the synthesis and implementation phase can take many hours, GBs of storage and RAM.

The final phase in the FPGA design flow is to generate a bitstream from the mapped and routed design and implement it onto an FPGA. At this point, the FPGA is ready to be used.

Each phase of the FPGA design flow is prone to contain errors and undesired behavior. As mentioned before the entire design flow can take a lot of time and resources. To detect possible design flaws and other errors at an early stage, verification and simulation are performed after each phase. After the Design Entry phase, behavioral simulation is performed to check the functionality of the HDL design agnostic of the type of FPGA the design is meant for. In this step, a designer can quickly identify functional or logical errors in the code and fix them.

After synthesis another functional simulation is used to once again verify the correctness of the netlist, but without taking the design constraints into account yet. The final simulation takes place after the Implementation phase. A highly detailed simulation is performed on the placed and routed design. Hardware design is a complicated process in which mistakes and errors are not easily detected. These simulations help designers to overcome these barriers when designing systems.

2.1.2. SoC FPGAs

Like ASICs, FPGAs are often used in a heterogeneous environment where they are responsible for handling critical parts of an application and work in harmony with processors. In such setups, communication between FPGA, memory, and processor is detrimental to performance. As the leading FPGA manufacturers identified this issue they introduced System on Chip FPGAs. A system where the processor and FPGAs are closely integrated, considerably decreasing the number of components and external interconnects [17]. To prevent ambiguity, the FPGA part of a SoC FPGA is called Programmable Logic (PL) and the CPU part including the main memory, is called Processing Subsystem (PS). In this work, the proposed solutions are developed for AMD SoC FPGAs. Therefore the tools and methodologies discussed will apply to AMD boards, it should be noted that similar tools and methodologies exist for Intel and other FPGA vendors. In specific, the experiments have been performed on low-end PYNQ-Z1 and high-end Alveo U280 FPGAs.

2.1.3. Platform resources

As previously mentioned, the general architecture of an FPGA consists of CLBs, I/O Blocks and interconnections resources. The fundamental components that are used to build a CLB are look-up tables (LUT) and flip-flops (FF). These components can be used to store data or implement logic. They can for example be used to define an operation or computation by implementing a truth table. Additionally, FPGAs incorporate different types of memory in the PL. The PYNQ-Z1 FPGA offers two different memory types, block random-access memory (BRAM) and look-up table random-access memory (LUTRAM), the latter is memory built from regular LUTs. The Alveo U280 FPGA contains an additional memory component, ultra random-access memory (URAM). BRAM and URAM are made up of a coarse-grained structure of larger memory blocks of 18Kb for the PYNQZ-1 and two 18, or one 36Kb block for the Alveo U280 [18]. LUTRAM consists of a more fine-grained structure, both boards have 6-input LUTs that can form different size memory blocks ranging from 64 to 512 bits. The previously mentioned memory components are in the PL of the FPGA. The PYNQ-Z1 SoC also contains 512Mb of DDR3 memory in the PS, which stands for Double Data Rate Synchronous Dynamic RAM (SDRAM). The stand alone Alveo U280 has 32 Gbs of on-board DDR4 memory as well as 8 Gb of High Bandwidth Memory version 2 (HBM2). Table 2.1 displays the resources available on the PYNQ-Z1 and Alveo U280 FPGAs.

	LUT (x1000)	FF (x1000)	DSP	BRAM	URAM	DDR (GB)	DDR total bandwidth (MBps)	HBM2 (GB)	HBM2 total bandwidth (GB/s)
PYNQ-Z1	**53.2**	**106.4**	**220**	**280 (18Kb)**	-	**0.5**	**1050**	-	**-**
Alveo U280	**1304**	**2607**	**9024**	**2016 (36Kb)**	**960 (288Kb)**	**32**	**19200**	**8**	**430**
Factor	**24.5**	**24.5**	**41**	**14.4**	**-**	**64**	**18.3**	**-**	**-**

Table 2.1: An overview of the available resources for the PYNQ-Z1 and Alveo U280 FPGA, The bottom row provides the factor of PYNQ-Z1 to Alveo U280 utilization. [11] [19].

The performance of an application on a SoC FPGA is not entirely dictated by the FPGA resources, the host processor contributes more or less depending on the proportion of the application that is executed outside of the FPGA. The PYNQ-Z1 platform is a SoC FPGA with an integrated Dual-core ARM Cortex-A9, the Alveo U280 is a standalone FPGA. The experiments in this work were performed on the Heterogeneous Accelerated Compute Cluster (HACC) at the ETH university in Zurich, Switzerland, where an Intel Xeon Gold 6234 processor is connected to the Alveo card through two 100 Gbps interfaces. Table 2.2 displays the specifications for the host processors of the targeted boards.

FPGA	CPU	Maximum frequency (MHz)	Cores	DDR3 (Mb)	DDR4 (Gb)
PYNQ-Z1	**ARM Cortex-A9**	**650**	**2**	**512**	-
Alveo U280	**Intel Xeon Gold 6234**	**4000**	**16**	-	**128**

Table 2.2: An overview of the available resources for the PYNQ-Z1 and Alveo U280 FPGA host processors [12] [20]

2.1.4. Direct memory access

Direct Memory Access (DMA) is a computer architecture component that allows direct data transfer between the PS and the PL. AMD provides users with pre-generated DMA IP blocks that handle read and write operations to and from different components. DMA on PYNQ-Z1 boards is used to transfer data through the advanced eXtensible Interface 4 (AXI4) interface between the host DDR and the kernels in the PL. The Alveo U280 utilizes DMA for several purposes. The DMAs can transfer data between the FPGAs DDR or HBM and the host DDR through a Peripheral Component Interconnect Express (PCIe) Gen3x16 interface. Additionally, the DMAs handle data transfer between the FPGA DDR and HBM and the kernels in the PL.

2.2. Frameworks and toolchains

In the following sections the different frameworks and toolchains used in this MSc. project will be discussed. AMD provides the user with several design tools and projects to target their FPGAs. These tools can be used in combination or standalone.

2.2.1. Vivado design suite

In 2012 AMD released the Vivado sesign suite, a software tool that enabled developers to analyze and synthesize hardware designs written in HDL such as Verilog or VHDL. In addition to its predecessor AMD ISE, Vivado introduces features that enable HLS and SoC development. An important characteristic of Vivado is that its purpose is to create hardware designs for PL. To create the software that interacts with the FPGA, AMD developed a different design suite on top of Vivado, called the Vitis unified software platform.

2.2.2. Vitis unified software platform

Vitis is a collection of software tools that together enable users to develop and exploit ready-to-use FPGA applications from a software oriented perspective. The platform consists of an extensive development kit, a growing ecosystem of hardware-accelerated libraries and other FPGA development related tools. Under the hood, Vitis uses the Vivado Design suite for the FPGA design flow. A relatively new, but key-feature of the platform is the ability to seamlessly duplicate implemented kernels in a hardware-design.

2.3. Scalability

Scalability can be defined by a system or application's ability to increase or decrease in performance and cost in response to changes in the processing demands [21]. In other words, how well can a system handle an increasing work-demand by adding resources to the system. An import aspect of the scalability of an application is the type and amount of parallelism that can be applied. In other words, in how many concurrent parts can an application be divided such that the parts can be processed by different nodes at the same time. There are two general types of software parallelism, data and task parallelism. This section illustrates both types of parallelism with the help of an example application containing 4 tasks that each take 1 second to execute. The dependency graph of the example application is show in Figure 2.3.

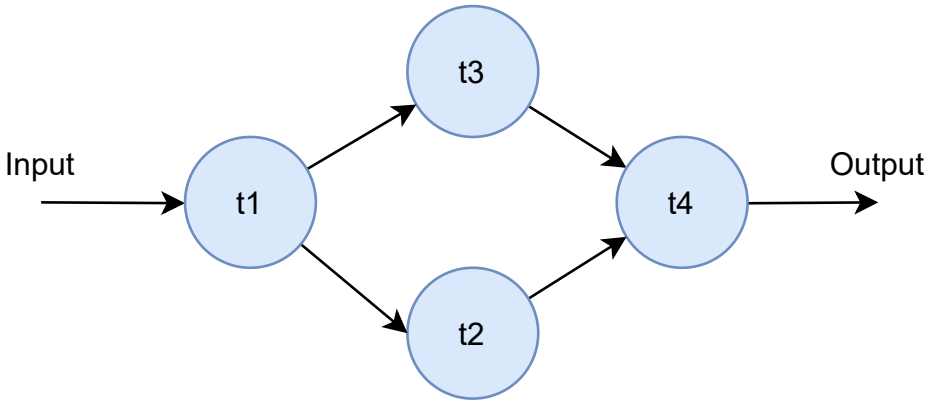

Figure 2.3: Dependency graph of parallelism example application.

The dependency graph shows that task 2 and 3 are dependent on the output of the first task and that task 4 is dependent of the outputs of task 2 and 3. If this application were executed on a sequential system where only one task can be processed at a time, it would take 4 seconds to execute.

2.3.1. Task parallelism

Task parallelism consists of dividing functional components of an application over multiple nodes such that they can be processed concurrently. The level of parallelism that can be achieved is strongly related to the inter-dependencies of the tasks. Task parallelism is applied on the example application and visualized in Figure 2.4.

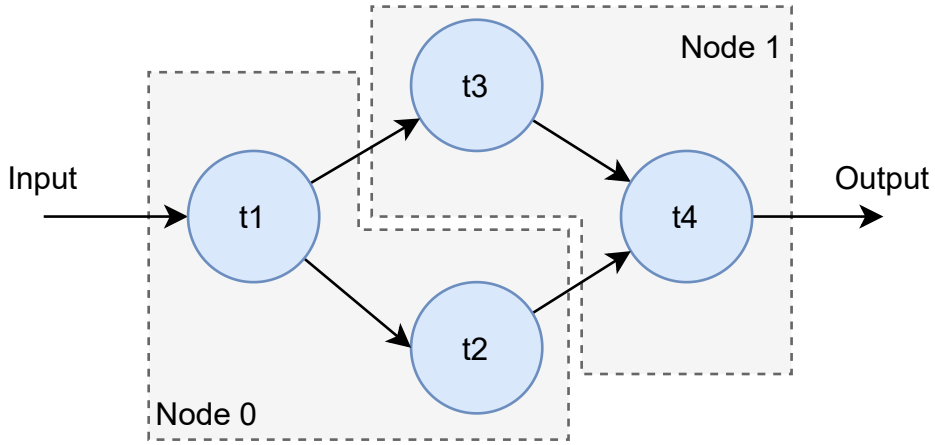

Figure 2.4: High level schematic of task parallelism.

Although the example application consists of four tasks, it only makes sense to scale it on two nodes because of the dependencies. Executing each task on a separate node would result in tasks 2 and 3 waiting for task 1 to finish, and task 4 on its turn waiting for task 2 and 3 to finish. Essentially only reducing the performance due to the overhead of data transfer between the nodes and increasing the cost by using 3 extra nodes. From the dependency graph it can be read that task 2 and 3 are the only two tasks that can be processed in parallel. Amdahl's law, shown in Equation 2.1, can be used to show the theoretically attainable speedup when parallelizing parts of an application. In the equation, $\mathtt{p}$ is the proportion of the application that can be parallelized and is scaled, $\mathtt{s}$ represents the speedup of the parallelizable proportion.

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.1}$$

Equation 2.1: A formulation of Amdahl's law [22].

In the example application, two of the four tasks are concurrently executed on 2 nodes. When applying $p = \frac{2}{4} = 0.5$ and $s = 2$ in Amdahl's law, shown in Equation 2.2, the total achieved speedup is 1.33 at the cost of one extra node resulting in an execution time of 3 seconds compared to the original 4 seconds.

$$Speedup = \frac{1}{(1-2) + \frac{0.5}{2}} \approx 1.33 \tag{2.2}$$

Equation 2.2: Amdahl's law applied to task parallelism on the example application.

2.3.2. Data parallelism

In data parallelism the same set of instructions or tasks are applied concurrently on different data. This is applied by splitting up the input data in several sets, processing each set concurrently and merging the results back together. Figure 2.5 demonstrates how this is applied to the example application.

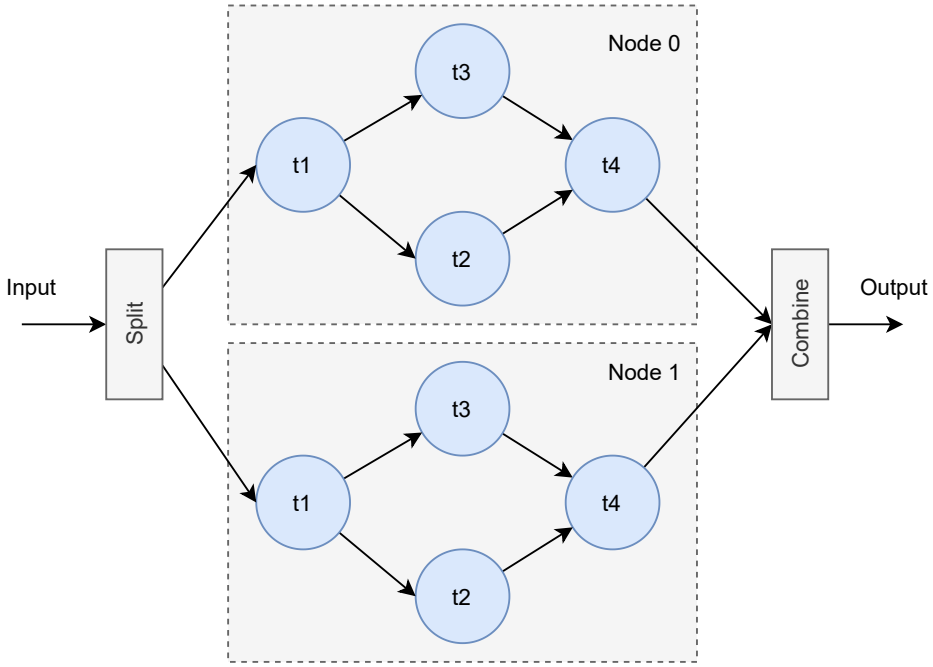

Figure 2.5: High level schematic of data parallelism.

Assuming the nodes execute the application in a sequential manner, the execution of the entire application takes 4 seconds. The application of data parallelism in this case does not increase the performance of the application, but rather allows the throughput to increase when processing the application multiple times. Two nodes can now process double the input size in the same execution time. Amdahl's law is applied on this example in Equation 2.3. As the entire application is parallelized over two nodes, $p = 1$ and $s = 2$, resulting in a speedup of 2 for double the input size.

$$Speedup = \frac{1}{(1-1) + \frac{1}{2}} = 2 \tag{2.3}$$

Equation 2.3: Amdahl's law applied to data parallelism on the example application.

By increasing the input size and adding more nodes, the speed up increases linearly as p remains the same and s increases i.e. 1000 nodes will process an input of 1000 times larger than the original in 4 seconds at the cost of 999 extra nodes.

Task and data parallelism are two constructs that can be used to scale an application. Task parallelism is often more complex to implement because it needs to manage task-dependencies as well as the data. Data parallelism is simpler concept to apply, but it requires an application for which the input can be split into different parts and processed independently. The example application demonstrates that data parallelism can theoretically be applied infinitely, being only bound by the input size and cost of nodes. Task parallelism can increase the performance of an application but its scalability is bound by the dependencies of the tasks. In practice, both types of parallelism can be used in combination on different levels to scale applications.

2.4. Related work

With the introduction of standardized accelerators such as the Alveo family, FPGAs have greatly risen in popularity. Leading chip-makers and cloud computing companies are respectively developing and exploiting FPGAs at an increasing scale. In addition to the commercial introduction of FPGAs, large operational clusters such as HACC [20] are available to the public. The barrier of FPGA design is being circumvented by the introduction of open source ready-to-use hardware libraries and slowly being tore down with the help of HLS programming tools [1] [16].

Additionally, frameworks are being developed to help integrate FPGAs in widely-used programming ecosystems. One initiative in this direction is PYNQ (abbreviated from Python Productivity for ZYNQ), an open-source project from AMD that makes it easier to use their FPGAs [23]. The framework was originally designed for the low-end ZYNQ SoC FPGA family, but now also supports Alveo cards and AWS-F1 instances. The integration of FPGAs and high-level language frameworks is not naturally efficient. For instance, Peltenburg et al. identified that the serialization bandwidth of high-level frameworks is an order of magnitude lower than modern FPGA interface bandwidth and developed Fletcher, a framework to efficiently integrate FPGA accelerators with Apache Arrow [5].

Other initiatives developed to increase the adoption of FPGAs are in the area of virtualization, allowing the accelerators to be exploited by multiple users in a cloud environment [24] [25]. A concept already widely applied for CPUs and GPUs.

Where virtualization allows multiple users to share FPGAs, an orthogonal research direction is aimed at using multiple FPGAs for a single task. Fleet, a framework developed by Thomas et al. [26], enables users to apply massive parallelization by duplicating FPGA implementations on multiple independent data streams. A newly developed HLS resembling language based on Chisel forms the foundation of the framework. A less sophisticated but much more accessible concept of duplicating FPGA implementations is incorporated by AMD in the Vitis and Vivado tools [27].

Due to its high computational and memory demand, Full Waveform Inversion is only recently becoming a viable seismic imaging technology. Large corporations such as Shell are working together with research groups to accelerate and improve implementations of the algorithm [28] [29]. As a highly parallelizable algorithm, research efforts show that FPGAs and GPUs can offer significant performance improvements compared to traditional CPUs [30] [31].

3

Application domains

This chapter introduces the applications used to answer the research question in this work, Full Waveform Inversion and Convolutional Neural Network. Sections 3.1.1 and 3.2.1 provide the reader with background information about the applications and the implementations used in this work. The sections conclude by characterizing the applications in terms of scalability in Sections 3.1.2 and 3.2.2.

3.1. Full Waveform Inversion

Full Waveform Inversion (FWI) is a data fitting method that aims to reconstruct velocity models of subsurface grids by iteratively solving a non-linear inversion problem. In this work, a linear approach is assumed to reduce complexity, the details of the linear approach are explained in Section 3.1.2. The non-destructive and high-resolution nature of FWI makes it suitable for applications such as seismic modeling and medical ultrasonic. Until recent years FWI was scarcely applied due to the heavy computational power required, the difficulty to estimate a good initial model, and the non-uniqueness of the solutions [31]. This section starts by providing background information about the algorithm and discusses some implementation specific aspects. Lastly, the application characteristics in terms of scalability are presented.

3.1.1. Background & implementation

Over the years, many FWI approaches have been studied to improve image quality and reduce computational complexity. Approaches mostly differ in the number of physical dimensions taken into account, the type of waves measured and assumptions on the background of the grid. In this work an approach developed by Peter Haffinger in his Ph.D thesis is used [32]. The approach implements FWI using acoustic waves in a 2-D model where a homogeneous background of the measured subsurface grid is assumed. Figure 3.3 shows a simplified diagram of the approach. Three stages are specified, the input, the iterative inversion and the output. The input consists of gathering the recorded pressure field data and making an initial estimate of the subsurface grid velocity model. The pressure field data is in practice obtained by physically recording it in a subsurface grid. A common method to obtain this data is by using sources, such as small explosions, to generate acoustic waves in the subsurface grid. For a defined range of frequencies, the pressure field of the reflected waves is captured by receivers such as microphones and serves as the observed input data. In Figure 3.1 an ideal case of 3D inversion is shown.

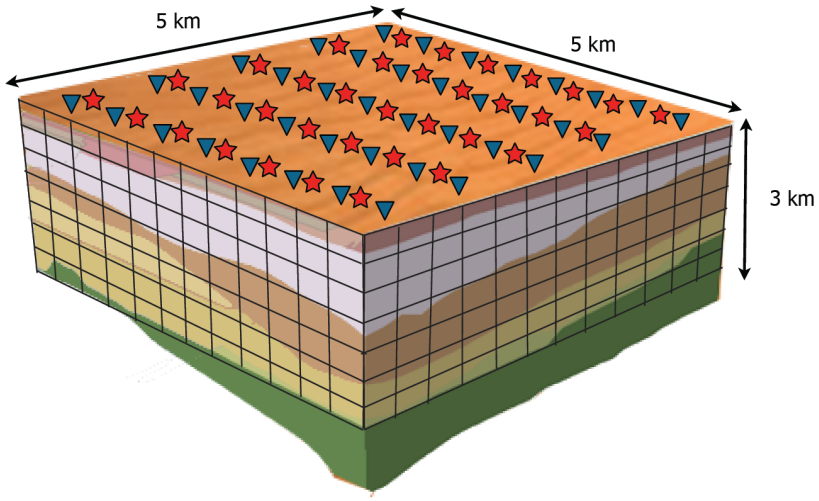

Figure 3.1: A high-level overview of the FWI algorithm.

In the figure, the surface is covered in a matrix of sources and receivers, marked as stars and triangles. The subsurface is discretized into a grid, where each cell represents a grid point. The approach in this work is based on several assumptions. First, the input measured data from the sources and receivers, called the observed data, is synthetically generated. As mentioned previously, a homogeneous background with a constant acoustic velocity of $c_0 = 2000m/s$ is assumed. The synthetic data represents layers of the subsurface structure with different, also homogeneous acoustics velocities. The contrast function of the acoustic velocities is in practice an unknown factor that is deduced from the seismic data. In this work, the contrast function, called chi (χ), is displayed in equation 3.1 where $c_0(\vec{x})$ represents the homogeneous background velocity and $c\vec{x}$ the subsurface layers' velocity.

$$\chi(\vec{x}) = 1 - \left[\frac{c_0(\vec{x})}{c(\vec{x})}\right]^2 \tag{3.1}$$

An example of a synthetic model is displayed in Figure 3.2, where on the left the original acoustic model is shown, the center represents the model reconstructed by applying FWI and the right shows the difference in chi between the original and reconstructed model. The horizontal and vertical axes represent the subsurface grid, a grid length of 1 in the figure represents 5 meters.

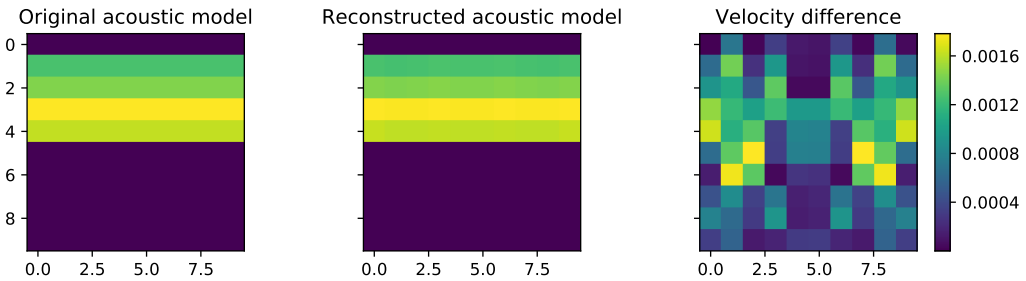

Figure 3.2: The synthetically generated acoustic model with a homogeneous background of $c_0 = 2000m/s$, embedded in the background are four layers with constant velocities from top to bottom of respectively 2200, 2233, 2300 and 2266 m/s. The differences are displayed by means of the contrast function and represent chi values. The grid is scaled where 1 grid length in the figure represents 5 meters.

The synthetic nature of this approach means the original acoustic model shown on the left of Figure 3.2 represents the true velocity model. To create the synthetic input data from this true model, the

pressure field data that is normally measured is calculated by solving wave equations of the acoustic waves generated and received from, respectively, the sources and receivers over a defined set of frequencies. This process is used extensively during the iterative stage of FWI and is called the forward model. The resulting pressure field data is a synthetic equivalent to the data measured in the field in the case of a real implementation. Because the approach assumes a homogeneous background, the initial estimation of the subsurface velocity expressed through the contrast function consists of zeros. In other words, the initial estimation of the subsurface grid's velocity is that it is equal to the homogeneous background at 2000 m/s.

The second stage of FWI consists of an iterative approach to improve the estimated velocity model until an acceptable difference in chi is attained, measured by a tolerance variable. Figure 3.3 shows the main steps that occur in the iterative inversion phase. The first step is to apply the forward model to the estimated velocity model to retrieve pressure field data. This data is then compared with the synthetically generated pressure field data. The difference between the calculated and synthetic pressure field data, called the residual, determines the quality of the current estimated velocity model. If the residual is larger than a predefined tolerance, the quality of the estimation is deemed inadequate. A cost function as part of an optimization technique is used to minimize the residual and update the estimated velocity model, this process is referred to as the inversion model. These steps are iteratively repeated until the residual is smaller than the predefined tolerance, resulting in a realistic model of the subsurface grid velocity. The output of the model is a reconstructed velocity model of the subsurface grid expressed with the contrast function chi. An example is displayed in the center of Figure 3.2. The implementation of FWI in this research, uses a Finite Difference Forward model and an inversion model based on the Conjugate Gradient Descent cost function. These implementations were chosen because they are less computationally intensive and highly parallelizable. An alternative for the forward model is the integral model. Alternatives for the inversion model are random inversion, evolution inversion and gradient descent inversion. A more in depth overview of the implementation can be found in Appendix A.3

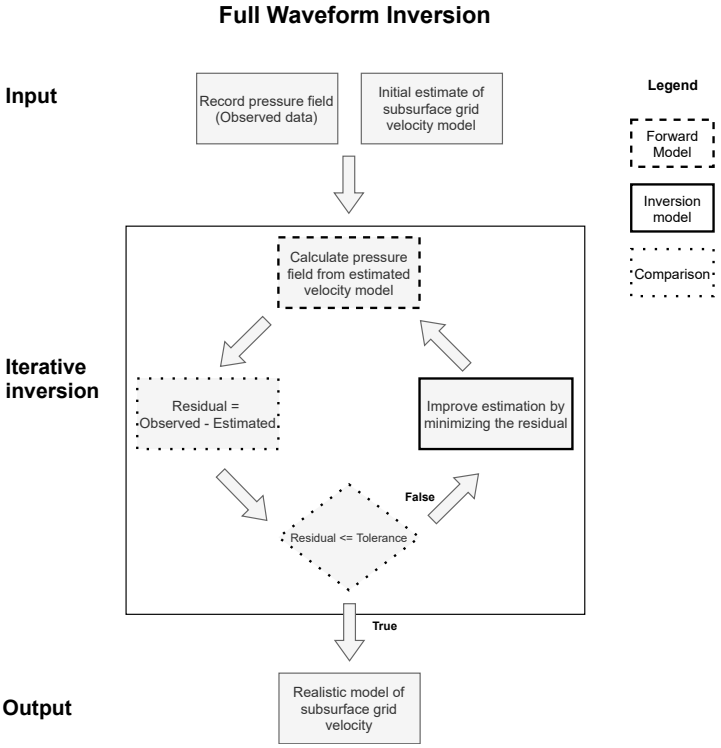

Figure 3.3: A high-level overview of the FWI algorithm.

3.1.2. Scalability characteristics

In real applications FWI needs to process incredible amounts of data. The inversion process is applied on a kernel further referred to as kappa. Kappa is built from a four dimensional function, the number of sources and receivers, the time measured and the number of unknown velocity values. For the sake of simplicity, in this work the time dimension is omitted. Before determining how kappa is built, an explanation is given of the linear and non-linear properties of FWI.

In FWI, each source produces a wave field that interacts with all the grid points in the subsurface grid. Every grid point that is not equal to the homogeneous background reflects a scattered wave field for every source originating field, effectively creating extra sources. The receivers measure not only wave fields from the original sources, but also from all the secondary sources. The interaction from the secondary sources causes a non-linear relationship between the actual velocities of the subsurface layers and the measured data. The kappa matrix represents the total field, where the pressure data for all the scattered and original wave fields for each measured frequency is mapped to the grid points. The dimensions of the matrix are $sources * receivers * frequencies$ rows and the number of grid points as columns.

This non-linear property forces FWI to use the entire input space of measured data to perform a realistic inversion. In a real application this input space is extremely large, an example based on the 3D model in Figure 3.1 is shown in Table 3.1. The example is based on a $5x5x3 = 75km^3$ domain where each cell is $5x5x5 = 125m^3$, this means there are $6 * 10^8$ unknown velocity values in the domain. By assuming a sample size of 8 bits the kappa kernel is $6.7 * 10^2$ Gb. These memory requirements become even larger if the time dimension is taken into account. The previously mentioned non-linearity results in the entire kappa matrix needed for the computations performed in the inversion model. Without quantum computers or much more powerful computing clusters, it is unfeasible to apply FWI for real 3D applications at this moment in time.

x	40000	# of sources
x	40000	# of receivers
x	$6 * 10^8$	# of unkowns
=	$8.3 * 10^{10}$	samples in kernel
x	8 bits	per sample
=	$6.7 * 10^2$ Gb	total kernel size

Table 3.1: Memory requirements for kappa kernel in a real application with 40000 sources and receivers, measured over a $5x5x3 = 75km^3$ grid with a cell size of $5x5x5 = 125m^3$ assuming 8 bits per sample. Example adapted from [32].

The FWI used in this work based on Peter Haffinger's thesis consists of 2D model, drastically reducing the kernel size. Another solution he proposed to reduce the complexity is to apply localization of the grid. Essentially ignoring the non-linearity of the entire subsurface grid in the local inversions, and only applying it after all the local domains have been inverted. In this work, the same concept of domain decomposition is applied to show FWI can be scaled. For the sake of simplicity, the non-linear conversion after inverting the local domains is not applied. It should be noted that the non-linearity due to wave field scattering inside the local grids is taken into account.

This work addresses the scalability of FWI by decomposing the subsurface grid into smaller domains. Additionally, the effect of a higher resolution, defined by the number of sources, receivers and measured frequencies, is analyzed. By ignoring the non-linear dependencies between the local grids, data parallelism can be applied by applying FWI independently on the local grids decomposed from the original subsurface grid. The parameters that have an effect on the computation and memory demands necessary to invert a local grid are the grid size and the resolution. The size of the kappa matrix can be used as a metric of complexity for FWI. The parameter decomposed by data parallelism is the grid size, which is only one dimension of the kappa matrix, the other three formed by the resolution. This results in an interesting characteristic when scaling FWI. Because the complexity of the problem size is dependent of the grid size decomposed by data parallelism and by the resolution, FWI is characterized as a multi-input dependent application. Consequences of this characteristic are that, although a very

large subsurface grid can be processed by applying data parallelism, the resolution forms the bounding factor of each nodes computational and memory requirements. In Chapter 6 a more in depth analysis of domain decomposition and FWI scalability is presented.

3.2. Convolutional Neural Network

A convolutional neural network (CNN) is a type of deep learning algorithm mostly used for image recognition and classification purposes. In recent years the quality of CNNs has drastically improved, sometimes even outperforming human accuracy [33]. Many research efforts were aimed at mapping NNs to FPGAs and ASICs of which a survey was published by Misra and Saha [34]. The FINN framework developed by AMD enables users to build flexible and fast quantized neural networks (QNN) hardware designs. The CNN discussed in this section is a binary NN (BNN) built with the FINN compiler.

3.2.1. Background & implementation

CNN fall in the category of supervised deep learning algorithms. As a form of an artificial neural network, the terminology used is based on the same concepts present in the human brain. Supervised learning entails that the algorithm is trained on a labeled data set, essentially learning what elements of an input correlate to a specific label. Once the training phase is complete and the weights and biases of the NN have been assigned, it can be used to classify new, unlabeled data. The process of making predictions, in the case of a CNN classification, is called the inference process. The CNN in this work is a trained algorithm that performs the inference process accelerated in hardware.

A CNN is a so-called multi-layer perceptron network where the neurons of the network are arranged in layers. The neurons from each layer take as input the output of all the neurons in the previous layer when fully connected. The data streams from neuron to neuron are called synapses, named after the circuit found in the human brain. During training, weights are bound to the synapses altering the inputs of a neuron and a bias is determined that is added to the sum of weighted inputs. Inside the neuron, an activation function performs a calculation on the weighted and biased input. The type of CNN discussed in this work is called binary CNN because the weights, biases, and activation functions are represented in single or double bits format instead of floating-point numbers. As a lot of the information held in floating-point numbers is redundant, BNNs are still able to achieve high accuracy while decreasing the complexity significantly [35].

A CNN topology typically consists of three types of layers, a convolutional, pooling, and fully connected layer. The convolution and pooling layers, often called hidden layers, are responsible for the feature extraction while the fully connected layer uses this information to classify the input image. The relationship between these layers is shown in Figure 3.4. A more in-depth explanation of these concepts applied to bins can be found in the FINN paper [35].

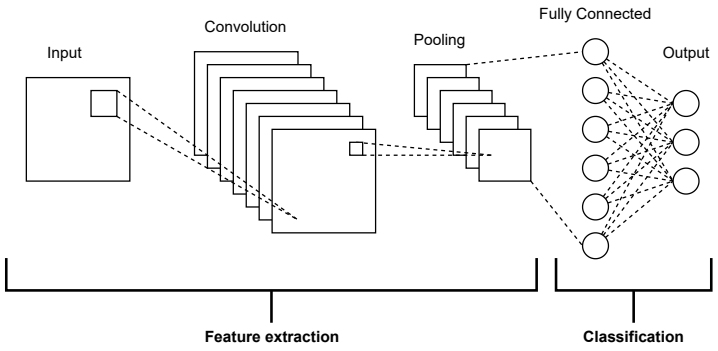

Figure 3.4: Visualization of the outline of a CNN, altered from [36]

The FINN framework [35] from AMD was used to build a binary CNN. The network is implemented to classify 32x32 RGB images. For this particular network, a one-bit activation layer and one-bit biases were used.

3.2.2. Scalability characteristics

A CNN classifies images one at a time in a sequential manner. As the input size of an image is fixed, the CNN takes approximately the same time to process each image. When applying data parallelism to scale a CNN, this means only one parameter is decomposed, namely the number of images. Unlike FWI, CNN can be characterized as a single-input dependent application where the problem size for each node is fixed.

4

Alternative solutions

Scaling data analytics applications on FPGAs require solutions on different layers of the stack. Figure 4.1 depicts an abstract representation of the four layers containing the proposed alternative solutions. This chapter consists of four sections that discuss the alternative solutions from the bottom to the top layer. At the end of each section, the proposed alternatives are compared in a qualitative table and then discussed. The last section of the chapter discusses the combinations of layer-specific alternatives and concludes with the chosen solutions. The following chapter proceeds to describe how the solutions are incorporated into the solution architectures and implemented.

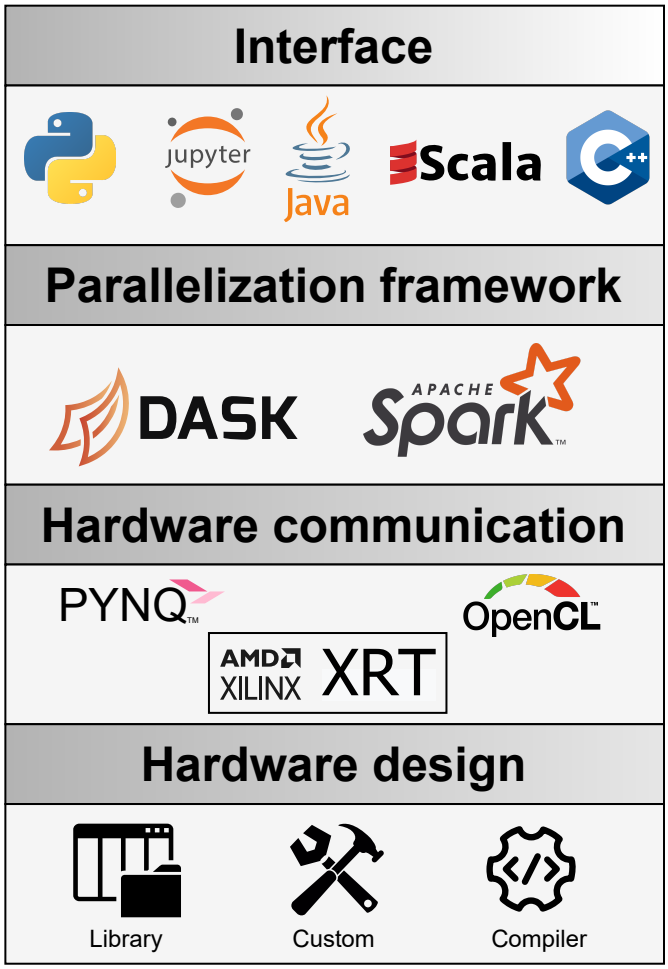

Figure 4.1: A high-level representation of the stack necessary to scale applications on FPGAs.

As seen in Figure 4.1, the lowest layer of the stack consists of the hardware design, programmed as a bitstream on the FPGA. This is where the hardware acceleration of the application takes place. The next layer is the hardware communication layer, an interface between the Programmable Logic and the software part of the application called the host code. The parallelization framework is the third layer, which is where the scalability of the application comes into play. The top layer is the interfacing language in which the host code is written. This layer is closely intertwined with both the hardware communication and the parallelization framework layer.

4.1. Hardware-design

The lowest layer of the stack consists of the FPGA hardware design, the core of the application acceleration. For this layer, we discuss three alternative methods presented below.

1. Custom hardware design.
2. Hardware designs from libraries.
3. Compiler generated hardware designs.

4.1.1. Custom hardware design

Developing custom hardware designs is a complex and time-consuming process that offers potentially high rewards. Experienced hardware engineers are able to express great control and precision in designs to generate optimal acceleration for any kind of application.

As explained in Section 2.1.1, the lowest abstraction level in the FPGA design flow is RTL where the hardware design is described in a Hardware Description Language. At this abstraction level, there are no specifications related to a type of FPGA board, rendering a vendor-agnostic and portable design. Although this approach offers many advantages in terms of flexibility, efficiency, portability, and performance, it is inaccessible to the general public.

4.1.2. Hardware designs from libraries

Historically, the electrical engineering (EE) community is not as accepting of the open-source environment the software community has embraced. In recent years, the openness of the software community is slowly being adopted in the form of open-source hardware libraries. AMD has launched an open-source platform for hardware-acceleration libraries, the Vitis Accelerated Libraries platform [1]. The platform is a collection of out-of-the-box acceleration libraries that offer the same level of abstraction as a software library would. These libraries are mostly developed for high-end FPGAs offering support for a large set of application domains such as data analytics, computer vision and data compression.

Specifically, machine learning and artificial intelligence have greatly risen in popularity in the data science domain. These types of applications are well parallelizable making them excellent candidates for FPGA acceleration. The Vitis AI development platform enables users to develop accelerated AI applications using popular frameworks like PyTorch, TensorFlow, and Caffe [37]. Vitis AI allows developers to exploit a well-optimized deep learning processing unit hardware design in their applications without the need for any hardware knowledge. A drawback of the Vitis Accelerated Libraries and Vitis AI is that they are bound to AMD-specific FPGA types and versions.

4.1.3. Compiler-generated hardware designs

A hardware design compiler partially or entirely automates the processes of the FPGA design flow described in Section 2.1.1. Most compilers, such as AMD and Intel's HLS tools, Matlab HDL coder, and many more, convert high-level code to an RTL written in HDL. Compared to using a hardware-library design, the compiler offers a user more flexibility to influence the generated design since the high-level code is custom written.

As explained in the previous section, Machine Learning and AI applications are well suited for FPGA acceleration. For that reason, many compiler projects were developed to compile different types of Neural Networks to hardware designs[38] [39]. The FINN compiler is one of the most advanced of these projects. It is an experimental framework from AMD that can be used to build custom Deep Neural Network (DNN) inference accelerators [35]. FINN is different from the Vitis AI platform, as it does not function as a generic DNN accelerator but rather offers the ability to explore design spaces of DNN inferences.

4.1.4. Evaluation

Table 4.1 displays a qualitative comparison of three methods to obtain a hardware design in terms of accessibility for users, configurability and portability of the design.

Method	Accessibility	Configurability	Portability
Custom	**- -**	**++**	**++**
Libraries	**++**	**-**	**+**
Compiler	**+**	**+**	**++**

Table 4.1: A qualitative comparison of different methods to acquire a hardware design.

FPGAs are suitable for many different application classes but the difficulty to engineer FPGA designs often forms a barrier hard to overcome for the general public. Hardware-design libraries and compilers are methods that allow hardware-inexperienced users to exploit the power of the accelerator. This ease

of use comes at the cost of configurability since libraries and compilers, to different degrees, provide a user with less control over the specifications of the design. Although being open-source, hardware libraries are often built for a specific vendor and board types. At the RTL abstraction level, designs are not attached to a specific vendor or board which makes the custom and compiler base designs much more portable over the range of market-available FPGAs.

4.2. Hardware communication

The hardware communication layer is where the application's host code and the hardware design interact. This layer of the stack is extremely vendor specific as it includes the drivers that communicate directly with the vendor-specific FPGA. Intel FPGAs use the Open Programmable Acceleration Engine framework to communicate with its devices, AMD provides the Xilinx Runtime Library (XRT) for this purpose. The experiments in this work were performed on AMD FPGAs. Therefore, in this section the focus lies on XRT and the affiliated APIs.

The Xilinx Runtime Library consists of four key functionalities listed below. The first two are necessary for the deployment and maintenance of FPGA accelerated applications. The latter two, if used correctly, can greatly influence the performance and scalability of applications.

1. Download of hardware design on FPGA
2. General board management
3. Execution management
4. Memory management

4.2.1. XRT

In Figure 4.2 an overview is given of XRT and the components it interacts with. XRT forms the bridge between the application host-code and the accelerator. The XRT core library consists of user-space libraries and Linux kernel drivers that are used as a multi-thread/process management system to perform the above-mentioned functionalities. On the accelerator side, static IPs such as Direct Memory Access (DMA) and Peripheral Component Interconnect Express (PCIe) (depending on the type of FPGA), control the accelerator's internal memory management. XRT interacts with these accelerator components through memory management drivers in the Linux Kernel.

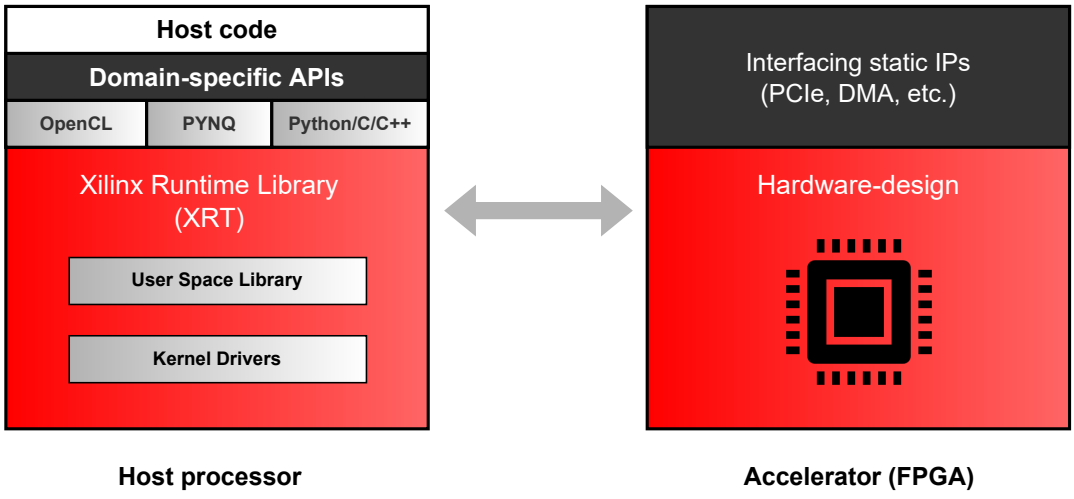

Figure 4.2: An overview of the Xilinx Runtime Library (XRT) adapted from [40]

On the host processor side, there are four different methods grouped in different levels of abstraction to interface with XRT, as shown in the top-left of Figure 4.2. The highest abstraction level allows domain-specific APIs such as TensorFlow and PyTorch from Vitis AI to incorporate XRT functionality. For a more specific and custom approach, XRT provides a native API, native libraries for C, C++, and Python, the PYNQ API, and OpenCL extensions.

4.2.2. OpenCL
OpenCL is an open standard for parallel programming in heterogeneous computing. There are several reasons why OpenCL is a good choice of API to interact with XRT. Primarily, the OpenCL standard is a well-known and developed API that many developers are familiar with. Moreover, the standard increases application portability because it supports a diverse range of accelerators such as GPUs, DSPs, and FPGAs.

4.2.3. PYNQ
The PYNQ API is built on top of the native Python bindings for XRT to provide a number of functionalities. Primarily, the 4 key functionalities of XRT are wrapped in a python interface enabling the same functionality as the native XRT libraries. As PYNQ was originally developed for the low-end ZYNQ platforms, the framework also provides libraries to interact with peripherals of the embedded devices such as UART, SPI, and IIC [23].

PYNQ can be used through an interactive, web-browser and OS-agnostic IPython environment in Jupyter notebook, or through a regular Python environment. This allows users to utilize python's rich ecosystem of libraries and APIs such as NumPy, OpenCV, and sci-kit-learn. The efficiency of C-code is generally higher compared to productivity-oriented Python code. PYNQ uses CPython and allows developers to use C-written libraries when efficiency is valued over productivity.

4.2.4. Evaluation
Table 4.2 displays a qualitative comparison of the methods to interact with XRT. The native XRT and native C, C++ and Python libraries are bundled under the name Native libraries as the metrics of comparison apply approximately equally to each of them.

Method	Accessibillity	Configurability	Portability
PYNQ	**++**	**+/-**	**+**
OpenCL	**+**	**+**	**++**
Native Libraries	**-**	**++**	**-**

Table 4.2: A qualitative comparison of methods to interact with the Xilinx Runtime Library (XRT).

The extra level of abstraction that OpenCL and PYNQ offer makes them considerably easier to use than the native libraries. Python is a more accessible programming language than C/C++ which is reflected in the accessibility score. Although a higher abstraction level results in better accessibility, it takes away some of the control and configurability the native libraries offer. PYNQ is a less mature and developed API compared to OpenCL directly influencing the extent of control the API currently offers. PYNQ can be used through an interactive, web-browser and OS-agnostic IPython environment in Jupyter notebook, or through a regular python environment. A disadvantage of PYNQ as well as the native libraries is that they are only compatible with AMD devices. OpenCL on the other hand can be used not only for FPGAs from any vendor, it also allows for portability across other accelerators like GPUs.

The choice of API to exploit XRT in the Hardware communication layer is closely related to the choice of the programming language used in the application host code. As will become clear in the following sections, the solution choices of the hardware communication, parallelization framework, and interfacing language layer are tightly intertwined.

4.3. Parallelization framework
The third layer in the stack facilitates the parallelization and scalability of the accelerated application. There are many different parallelization frameworks available, in the scope of this work two widely used frameworks are discussed, Apache Spark and Dask.

4.3.1. Apache Spark

Apache spark is a well-developed and professionally established framework written in Scala. The framework was extended to also support Python and R but the compatibility with these languages is not as fluid as with the intended languages Java and Scala. Spark is built on the concept of the Hadoop ecosystem and uses the Resilient Distributed Dataset (RDD) paradigm in combination with lazy evaluations and in-memory caching to deliver low-latency scalability for distributed big data tasks. Working with Spark can be a challenge for new users as the RDD paradigm, Hadoop ecosystem and Spark way of working are not trivial to learn. [41]

4.3.2. Dask

A pure Pythonesque alternative parallelization framework is Dask. Dissimilar to Spark, Dask was built with the intention to smoothly integrate with python allowing users to scale their existing applications with minimal effort. As dask is a younger and open-source project, there is less commercial support available compared to Spark. Dask allows users to scale their applications in two ways:

1. On (heterogeneous) multi-node clusters.
2. Locally, on a single machine where it leverages multi-core CPUs and efficient data handling.

Dask consists of three classes of modules that together enable many different configurations of parallelization and scaling. Figure 4.3 shows the different classes of modules and how they relate to each other. Collections are modules that define how data is bundled. Dask supports collection modules for many data-oriented Python APIs such as NumPy, pandas, and sci-kit-learn as well as native data collections called dask bags.

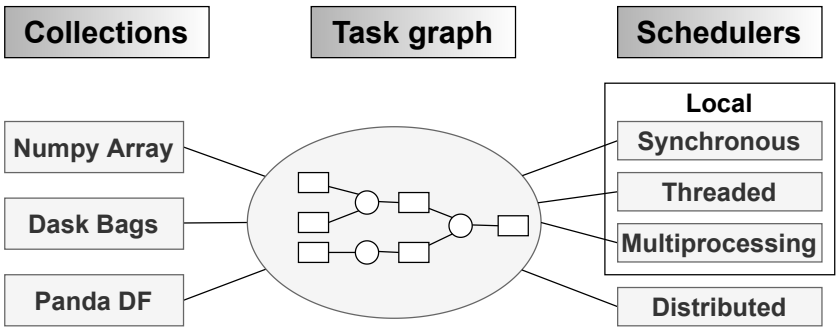

Figure 4.3: The different classes of Dask modules and how they relate, figure adapted from [42]

On the right side of Figure 4.3 are different types of scheduler modules. Schedulers are responsible for executing Directed Acyclic Graphs (DAG) called Task Graphs, which consist of operations or computations on the collections. The dask distributed scheduler allows parallelization and scaling over multi-node clusters through a TCP connection. It consists of a central dask-scheduler, a dask-client, and an arbitrary amount of dask-workers spread over the cluster.

4.3.3. Evaluation

Table 4.3 displays a qualitative comparison of the discussed parallelization frameworks. Both frameworks offer well-developed kits that excel in terms of scalability and performance when used correctly. Dask outscores Apache Spark in accessibility as the latter requires slightly more initial effort from the user to work with efficiently.

Framework	Accessibility	Performance	Scalability
Apache Spark	**+**	**++**	**++**
Dask	**++**	**++**	**++**

Table 4.3: A qualitative comparison of parallelization frameworks Apache Spark and Dask.

4.4. Interface

The interface forms the programming language in which the host code of the application is written. In addition to the core functionality of the application, the interfacing language is used to interact with the parallelization framework and the hardware communication layer.

4.4.1. Programming languages

Nowadays, there are hundreds if not thousands of programming languages available to code applications in. Different aspects are important when considering the correct language for an application, and often there are multiple viable solutions. The PopularitY of Programming Language Index (PYPL) [43] represents the popularity of programming languages based on the number of google searches for tutorials in the language. The top 6 languages from June 2022 are shown in Table 4.4 where the languages marked with a star are the most used and applicable in the data science community.

Rank	Language	Share (%)
1	Python *	27.61
2	Java *	17.64
3	JavaScript	9.21
4	C#	7.79
5	C/C++ *	7.01
6	PHP	5.27

Table 4.4: The most popular programming language from [43] * marked languages are the most popular in the data science community

In the scope of this work, only programming languages that are commonly used for data science and analytics are taken into account. Considering those, Python, Java, and C/C++ are possible alternatives for the interfacing language.

4.4.2. Evaluation

Table 4.5 gives a qualitative comparison of the different interfacing languages for the metrics important to this work: accessibility, performance, and scalability.

Language	Accessibility	Performance	Scalability
Python	**++**	**+**	**++**
Java	**+**	**+**	**++**
C/C++	**-**	**++**	**-**

Table 4.5: A qualitative comparison of the different interfacing languages based on accessibility, performance and scalability.

Python is an easy-to-learn extremely versatile functional language with a rich ecosystem of libraries that makes it, amongst other things, well suited for data science applications. As can be seen in Table 4.4, Python is the most popular language at the time of writing. This popularity impacts the accessibility of the language through the largest community, documentation, and support available online. Java and Scala are a part of the Java Virtual Machine (JVM) ecosystem. Like Python, Java is one of the most popular languages with a similar amount of resources available online. The English resembling the intuitive syntax of Python is the reason it outscores Java in terms of accessibility. Although C/C++ are also well documented, they are considered much harder to learn than Java or Python as they also require knowledge of manual memory management and other low-level concepts. The low-level aspects of the C and C++ languages give developers great control on how to optimally write code for the underlying hardware to optimize performance. Scaling over distributed systems is a relatively new technology that has mostly evolved in newer languages like Java and Python. Because the languages handle more low-level tasks automatically it is easier to seamlessly scale applications over multiple machines, where C/C++ applications often require more manual effort.

4.5. Conclusion

The previous sections of this chapter have described and compared alternative solutions for each layer of the stack. The evaluations of each layer were used to construct three solution architectures for the implementation of the two chosen applications, Full Waveform Inversion, and the Convolutional Neural Network (CNN). The solution architectures discussed one by one in the following sections are visualized in Figure 4.4.

4.5.1. FWI on PYNQ-Z1

The first solution architecture is designed to implement FWI on a PYNQ-Z1 FPGA. The FWI application is a custom-written algorithm for which no hardware libraries are available. The hardware design was therefore created with the help of HLS tools in combination with custom implementations. The targeted board dictated the choice of hardware communication layer to be PYNQ. The logical choice of parallelization framework then becomes python-oriented Dask, allowing for smooth integration with also python-based PYNQ. The implementation of the FWI algorithm provided by ALTEN was written in C++. Although sacrificing some performance when implementing the algorithm in python, the seamless integration with Dask and PYNQ allow the user to scale FWI on multiple boards.

4.5.2. FWI on Alveo U280

FWI was also implemented on the high-end Alveo U280 FPGA. This solution architecture is twofold, as a version with OpenCL as well as PYNQ was implemented. The hardware design was custom built with the help of HLS tools for the same reason as described in the previous solution architecture. As the original algorithm was written in C++ and the targeted platform does not require the use of PYNQ, OpenCL was used to create a high-performing single-node solution. In the discussed solution, Java integrates well with the parallelization framework Spark but does not have an equivalent to PYNQ for hardware communication. With the lack of a parallelization framework for C++, this solution architecture also used the combination of Dask and PYNQ to scale FWI on multiple Alveo U280 nodes.

4.5.3. CNN on Alveo U280

The last solution architecture hardware design was generated with the FINN compiler. The host code of the application, taken from an AMD repository, is written in python and interfaces with PYNQ. Again the logical choice to scale this application became the Dask parallelization framework.

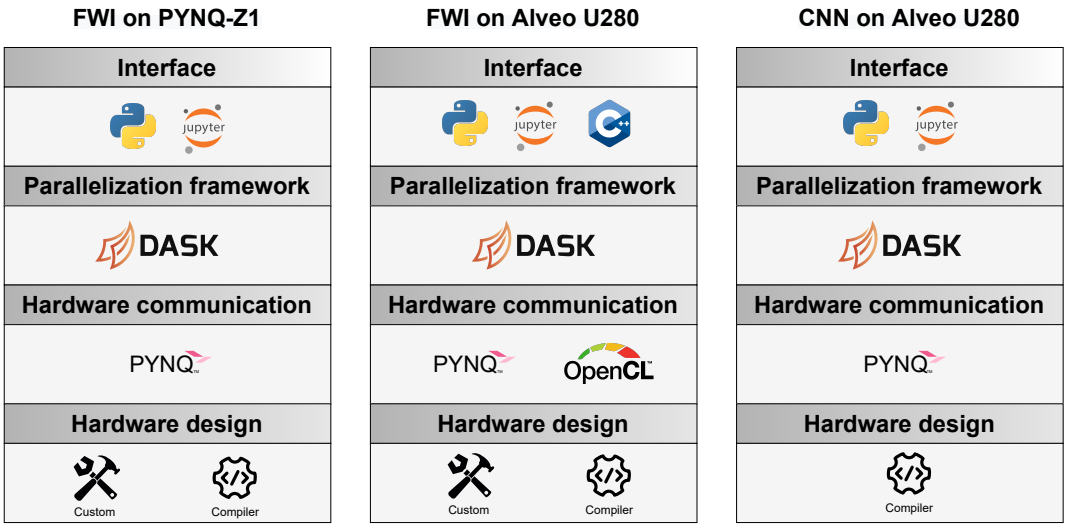

Figure 4.4: A visualization of the solution architectures for FWI on Alveo U280 and PYNQ-Z1 and CNN on Alveo U280.

5

Solution architectures and implementations

As there are redundancies present in the solution architectures, this chapter discusses the individual components used to build them. The chapter begins by presenting the implementation of the hardware design for FWI in Section 5.1. The CNN application was compiled with the FINN compiler, the section sends by presenting the resource utilization of the CNN hardware design. Having established the hardware designs for FWI and CNN, the implementation of the hardware communication layer with PYNQ and OpenCL is presented in Section 5.2. Additionally, this section provides an explanation how to create hardware designs with multiple instances of kernels. The chapter finishes with an extensive description of the old and new OctoRay framework in Section 5.4.

5.1. FWI hardware design

In this section, the implementation of the hardware design for FWI is presented. Section 5.1.1 analyzes what components of the FWI algorithm are suited to be accelerated and incorporated into the hardware design. Successively, Section 5.1.2 describes the suitability for the acceleration of the selected components. The initial implementation of the decided-upon kernels is presented in Section 5.1.3 succeeded by the optimizations leading to the final design in Section 5.1.4.

5.1.1. Algorithm analysis

Full Waveform Inversion required a custom implementation of the hardware design. An investigation was performed to analyze the behavior of different code sections of the algorithm by scaling the input parameters and analyzing the execution time per section. The execution time until convergence is influenced by the four dimensions of kappa, mentioned in Section 3.1.2 and the tolerance level at which an acceptable reconstructed model is accepted.

For the investigation, the tolerance was kept constant at the default value of $10 * 10^{-6}$, this value represents the difference between the original and reconstructed model based on the contrast function, chi. The number of frequencies, sources, and receivers multiplied by each other represents the resolution of the algorithm. In the ALTEN implementation, the number of sources and receivers are required to be the same. The number of frequencies is an independent parameter that can differ from the number of sources and receivers. For this investigation, the number of sources, receivers, and frequencies and the grid dimensions are scaled near equally to minimize the effect of the individual parameters on the computational load. The impact of these independent parameters on the algorithm is outside the scope of this work as it is dependent on geophysical properties. This investigation was performed on a system with an Intel I7-8750h CPU containing 12 logical cores running at 2.20 GHz and 16GB of RAM.

Figure 5.1 shows the percentage of the total run time spent in sections of the algorithm. Two separate code sections, used in multiple parts of the algorithm, were quickly identified as the most expensive.

For the sake of clarity, the two code sections are specified as the Forward function and the Cost function, which represents the overarching functionality these sections reside in. The combined function's parameter represents the sum of these functions. The Iterative loop parameter represents the time spent in the core of the algorithm, iterating over the forward model and inversion model until the algorithm converges.

A number of conclusions can be made from Figure 5.1. The part of the total run time that is not incorporated in the Iterative loop parameter consists of the initialization and wrap-up of the algorithm. As the grid size increases, it stabilizes at approximately 10% of the total run-time, nearly unchanging with respect to the resolution. This indicates that the resolution only slightly affects the initialization and wrap-up phases and that these phases grow linearly with the grid size.

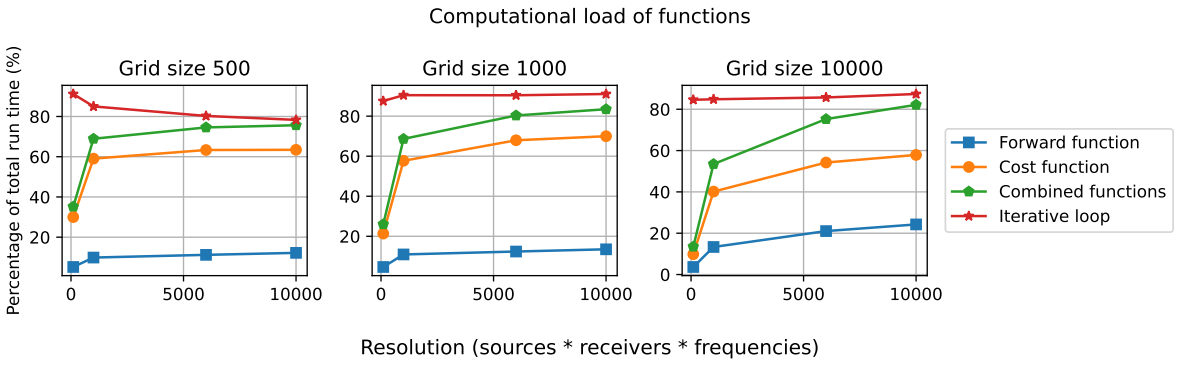

Figure 5.1: An overview of the percentage of run time occupied by the forward, cost, and combined functions and the iterative loop for grid sizes 500, 1000, and 10000 and resolutions (sources*receivers*frequencies), $100 = (5 * 5 * 4)$, $1000 = (10 * 10 * 10)$, $6000 = (15 * 15 * 20)$ and $10000 = (20 * 20 * 25)$.

The trends of the Cost and Forward functions are approximately the same when scaling the grid size. The percentages are the most sensitive to the increase in resolution, which makes sense as the resolution forms three of the four dimensions kappa consists of. The purpose of this investigation is to decide on the sections of code that will decrease the execution time the most when accelerated in hardware. As grid size and resolution increase, the combined functions take up nearly all of the execution time in the Iterative loop. At first glance, these functions seem the best choice to accelerate in hardware. Before committing to a hardware design, further analysis was necessary to define the parallelization opportunities and acceleration fitness of these functions. This analysis is presented in the following sections where a high-level overview of the two functions is given.

5.1.2. Hardware kernels

In Chapter 3, the forward model was introduced as the FWI component used to calculate the pressure field from the estimated velocity model. One of the operations to achieve this is applying the kappa matrix to the current estimation of the velocity model by performing the dot product of each row with the current estimated pressure field data vector. The vectors in the kappa matrix and current estimated pressure field data contain complex numbers. Performing a dot product of vectors for all the rows is a highly parallelizable operation which makes the Forward function an excellent candidate to be accelerated in hardware. The C++ code of this component of the forward model is shown in Appendix A.2.

The cost function used in this FWI implementation is the Conjugate Gradient Descent function. The goal of this function is to numerically solve a system of linear equations. In FWI that means minimizing the residual between the observed and calculated pressure field data. In order to find the optimal solution, in each iteration, a direction in the linear space is identified and step size is calculated. The Cost function specifies the construct to calculate the updated direction. In this function, the residual vector containing floating-point numbers is applied to the kappa matrix. Like the Forward function, the cost function is a good fit for hardware acceleration as the operations performed are vector operations paral-

lelizable over the rows of the kappa matrix. The C++ code of the cost function is shown in Appendix A.1.

5.1.3. Design implementation

The FWI algorithm analysis led to the decision to create a hardware design consisting of two kernels, one for the Forward function and one for the Cost function. This section first presents the implementation of the hardware design for the PYNQ-Z1 SoC FPGA as this required the most manual approach. The Vitis toolchain used for the Alveo U280 implementation automates a number of steps in the design flow. The Alveo-specific implementation steps are discussed at the end of the section.

The kernels for PYNQ-Z1 were designed with the Vivado HLS tool. The IP blocks generated with HLS were then linked together to a functional design in Vivado Design Suite. The design flow for this process consisted of multiple iterations to achieve the final, optimized result.

Developing a kernel with an HLS tool consists of several steps. Firstly, the functions are extracted from their context in the algorithm and coded as separate, independent functions. The next step is to code a test bench that can validate the correctness of the extracted functions. Once the code has been validated it needs to be adapted to handle the data transfer between the FPGA and host code. For this purpose the DMA IP blocks are used, that can directly access the main memory without having to pass through the CPU. Vivado HLS provides the `hls::stream` wrapper to create a data-stream between the main memory and FPGA. The header of the Forward Function where this has been applied is shown in Listing 5.1.

Listing 5.1: Header of the Forward function kernel

```
1 void ForwardFunction(hls::stream<std::complex<float>> &kappa_matrix,
2     hls::stream<float> &cur_est_pressure_field,
3     hls::stream<data_struct<std::complex<float>> > &output_vector)
```

The `output_vector` is wrapped in a struct that consists of the output vector and an integer, named `last`, that is necessary to inform the DMA when the last value of the vector was streamed back to the main memory and the transfer can be closed. The test bench is also adapted with these changes and the result is used to validate the FPGA-ready kernel code. Finally, the HLS tool is used to synthesize and implement the validated kernel code into IP blocks for the selected hardware platform, the PYNQ-Z1 board.

As explained in chapter 2, this hardware platform is an SoC FPGA consisting of a Processing System (PS) and Programmable Logic (PL). In the Vivado Design Suite, the HLS-generated IP blocks are imported. They are linked together with other relevant components such as AXI DMAs in the PL. Additionally, the settings of the DMAs such as they write and read streams, signal properties, stream, and memory map data width, etc. are configured. The specified data widths of the AXI DMA interfaces need to match the size of the data types that will flow through the streams. The Cost and Forward function inputs and outputs are of data type `complex<float>` or `float` resulting in data widths of respectively 64 and 32 bits. Each DMA stream is configured according to the attached input or output data type and its function as the Main Memory to Stream (MM2S) or Stream to Main Memory (S2MM) interface. Finally, the Zynq Processing System IP is connected to the design. This IP block is a wrapper that serves as a connection between the PL and PS. The resulting block design, shown in Figure 5.2, is a functional but not yet optimized design that is ready to be synthesized, implemented, and turned into a programmable bitstream.

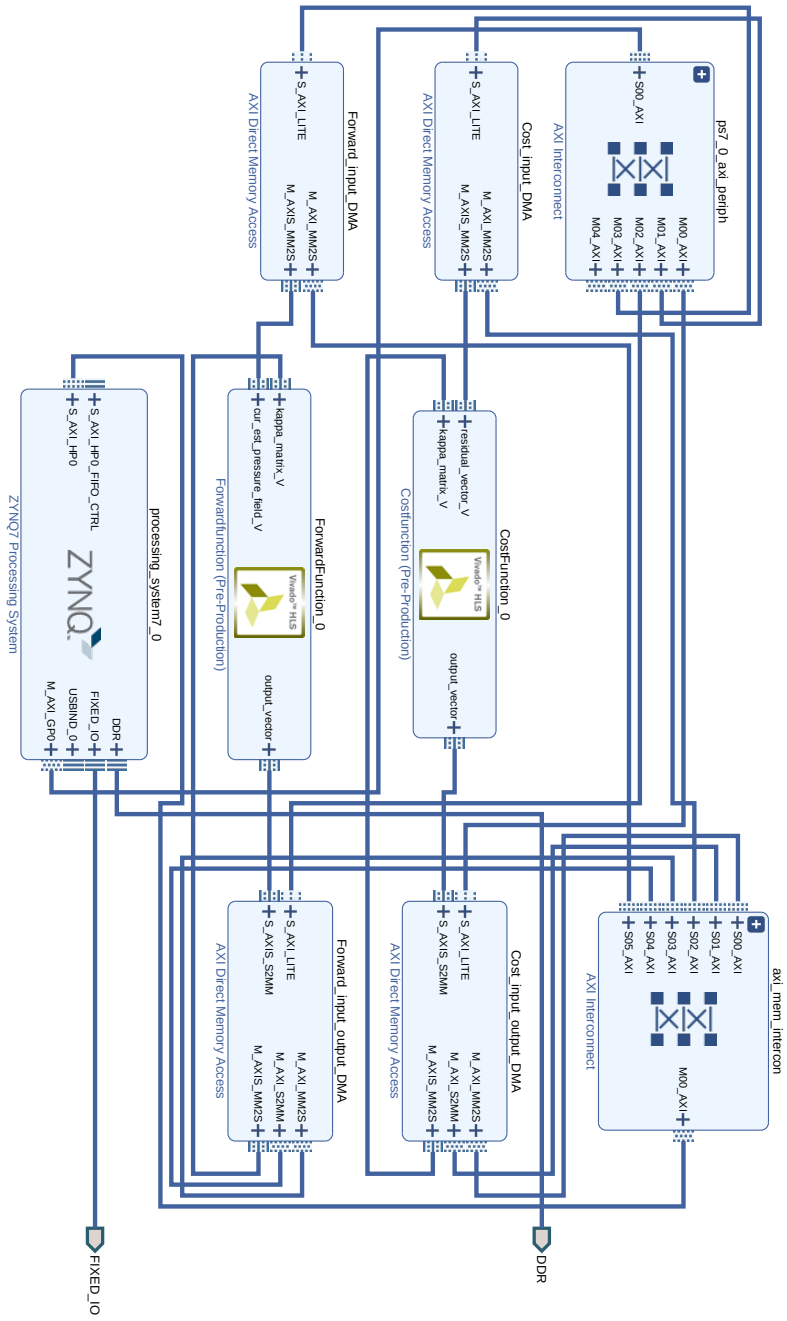

Figure 5.2: The Zynq Processing System IP, taken from [44]

The Alveo U280 implementation requires the Vitis platform to be used for the design flow. The platform incorporates Vivado and automatically handles many of the lower-level design aspects such as the linking of IP blocks and DMA configurations. Consequently, implementing the hardware design from Alveo platforms simply consisted of configuring the correct C++ compiler for the host code and specifying the memory banks the kernels should be connected to. All the other implementation steps are taken care of automatically by Vitis.

5.1.4. Design optimization

The design optimizations in this section were initially performed for the PYNQ-Z1 FPGA, a hardware platform with relatively few resources available. As the optimization steps were almost identical for the Alveo U280 implementation, unless otherwise indicated, the optimizations described in this section are also applicable to the Alveo U280 implementation.

For the initial design of the Forward and Cost functions, a small input size was used to guarantee the design would not exceed the resources available. In the design optimization phase, multiple steps were taken to improve the design and maximize the utilization of the FPGA resources. HLS tools provide pragmas to optimize the design by reducing latency, increasing the throughput, and better utilization of the resources.

The Forward and Cost kernels have a similar code structure where operations are performed between an input vector and the kappa matrix. To achieve the highest parallelization degree, the entire kappa matrix and input vector need to be contained in an FPGA memory component. In other words, the resolution and grid size that respectively represent the number of columns and rows of the kappa matrix dictate the memory requirements. The Forward function consists of a dot product between the kappa matrix and the currently estimated pressure field vector, a simple calculation that can be implemented using relatively few FFs and LUTs. The Cost function applies a similar low resource set of operations on the kappa matrix and residual vector. From this characterization, we can conclude that when scaling the size of the kappa matrix and input vectors, memory resources will form the bottleneck.

Two optimizations techniques were applied to improve the memory resource utilization by using HLS pragmas `array_partitioning` and `resource`, the most optimal designs per optimization level are displayed in Table 5.2. Additionally, the `pipeline` pragma was used to ensure the concurrent execution of the loops. The initial design before optimizations fit an input resolution of 125 and a grid size of 100. The following sections first discuss how the HLS memory-related pragmas were used to improve memory utilization and consequently increase the manageable resolution and grid size. Secondly, the effect of the `pipeline` pragma to enable concurrent execution of the kernel code is discussed.

Array partitioning

Array partitioning is the process of splitting up an array into smaller arrays. There are three forms of array partitioning available in Vivado HLS, complete, block, and cyclic partitioning [45]. The default type is complete partitioning, where the array is decomposed into individual elements. In a multi-dimensional array, each dimension forms an element, in the case of a one-dimensional array, each element is stored in an individual register. Figure 5.3 visualizes block and cyclic partitioning for an 8-element array and a factor of 2. Block partitioning decomposes the array into consecutive blocks of equal size. By using the `factor` argument, the number of blocks and sequentially the block size are configured. In cyclic partitioning, the `factor` argument specified the number of arrays the original array is partitioned over. A factor N means that the first N elements of the original array are chronologically assigned to the first index of the N arrays. The following N elements will be assigned to the second index of the arrays, this cyclic process continues until the original array is completely partitioned.

Cyclic partitioning showed the best results of the three techniques, an example of the use of the array partitioning pragma is shown in listing 5.2. This optimization allowed for a design with nearly twice the input size to be generated. The resources of the initial and optimized design are displayed in Table 5.2.

Listing 5.2: Example use of the array partition pragma for a 2 dimensional array with cyclic partitioning using factor 20.

```
1 #pragma HLS array_partition variable=kappa_matrix cyclic factor=20 dim=2
```

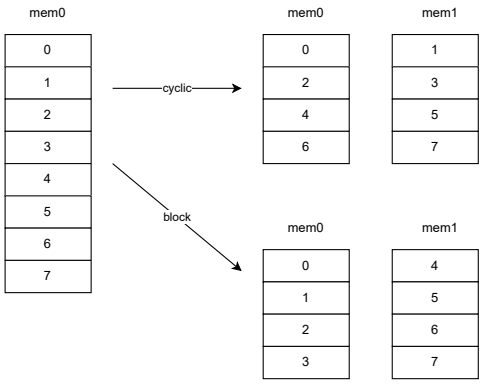

Figure 5.3: Visualization of block and cyclic array partitioning of an 8-element array with factor 2, taken from [46]

Resource

After applying array_partitioning the design was able to manage a significantly larger input size. The utilization numbers of the design show that 89% of the BRAM is used while the other resources are still available in abundance. The HLS tool by default utilizes BRAM as the memory component. The tool does apply some optimizations by default, it can be noted that by introducing array_partitioning the LUTRAM and DSP utilization increased without any explicit user specification. The `resource` pragma allows a user to allocate memory to a specific memory type of which an example is shown in Listing 5.3 [47]. To implement this, in each design the kappa matrix was decomposed into two different size arrays. The size proportion of the BRAM and LUTRAM arrays was configured to maximize the utilization of both resources.

Listing 5.3: Example use of the resource pragma with 1 port LUTRAM..

```
1 #pragma HLS resource variable=lutram_kappa_matrix core=RAM_1P_LUTRAM
```

The `resource` pragma is called `bind_storage` in the Vitis toolchain. Additionally, as mentioned in Section 2.1.3, the Alveo U280 contains another memory type called URAM. This memory type was primarily used for the Alveo U280 implementation as there is 35Mb of URAM and only 4.5Mb of BRAM available on the board.

Pipeline

The previous optimization techniques were used to improve memory utilization. A third optimization was performed to improve the latency of the design. Both the Cost and Forward function kernels consist of a number of for loops in which operations are performed. HLS provides the pipeline pragma to allow concurrent execution of the operations which reduces the initiation interval (II) of the loop. The II is the number of clock cycles between each new input the loop can process, by default the pragma sets the II to 1. If the specified II can not be achieved, the HLS tool increases the II until a viable interval was found. An example use of the pragma is shown in Listing 5.4.

Listing 5.4: Example use of the pipeline pragma with the default II of 1.

```
1 for (int row = 0; row < LOW; ++row){
2 #pragma HLS PIPELINE
3     sum += kappa_matrix[row][col] * cur_est_pressure_field[col];
4 }
```

Figure 5.4 demonstrates how pipelining a loop allows for concurrent execution of its operations resulting in reduced latency. In this example, the II is 3 clock cycles, without pipelining the execution of the loop until the last output write, which is 8 clock cycles. By concurrently executing the operations in a pipeline it is reduced to 4 cycles. As displayed in Table 5.1, applying pipelining to the Cost and Forward kernels reduced the latency respectively by a factor of 9.37 and 2.95.

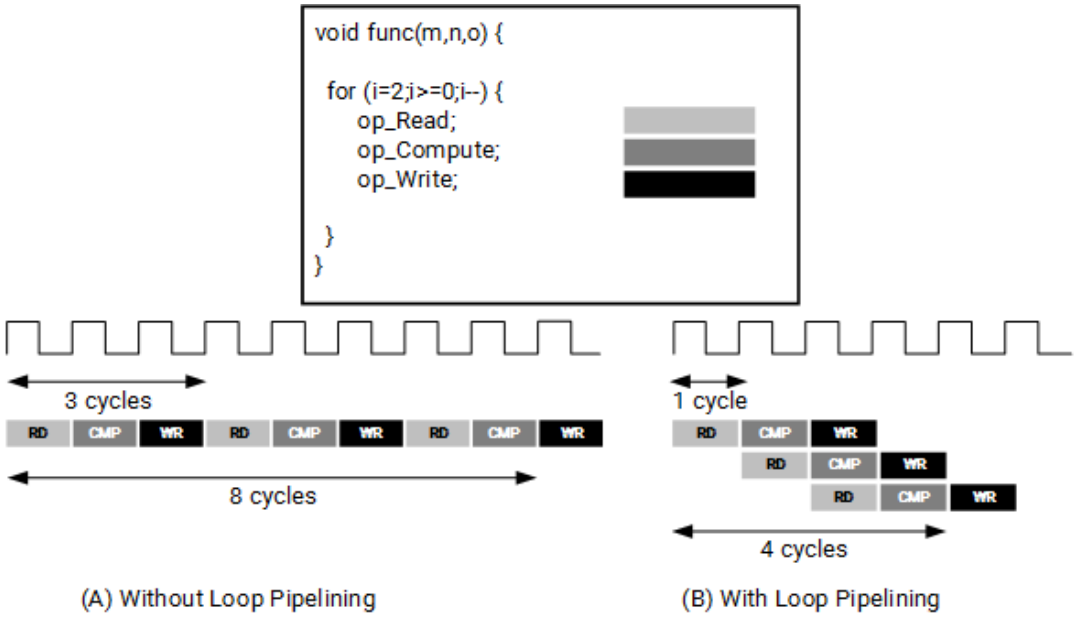

Figure 5.4: A visualization of the effect of pipelining, taken from [48].

Another effect of pipelining is the increase in operational clock frequency. In the example from Figure 5.4, sub-figure (A) processes an output every 9 clock cycles, in this example the last write operation is also taken into account. If a clock period of 1ns is assumed, processing an output takes 9ns (1ns * 9 clock cycles). The 9 cycles form the longest and thus critical data path in this design, dictating a minimal operating clock frequency of 111.11MHz (1 / 9ns). In sub-figure (B) the pipeline is filled after 3 cycles. From that moment, every clock cycle and output is processed, dictating a minimal operating frequency of 1000Mhz (1/1ns), a 9-fold increase compared to the sub-figure (A). Surprisingly, it can be read from Table 5.1 that the estimated clock period increases, and consequently the frequency theoretically decreases when applying pipelining. There are many factors that affect the clock period of a design, pipelining makes the design more complex and increases the area utilized. A larger area and complex design with longer data paths often result in higher clock periods. Applying pipelining to the Cost Function increased the utilization of the BRAM by 0.7%, DSP by 3.7%, FF by 1.4%, and LUT by 8.5%. Investing these resources in increasing the resolution and grid size input would only result in minimal improvements. When pipelining, the reduced latency significantly improves the throughput of the design, outweighing the minimal gains when investing the resources in improving memory utilization.

Kernel	Optimization	Absolute latency (ms)	Estimated clock (ns)
Cost Function	**Not pipelined**	5.671	7.256
	Fully pipelined	0.605	8.946
Forward Function	**Not pipelined**	5.415	7.256
	Fully pipelined	1.836	8.332

Table 5.1: Latency improvements when pipelining the PYNQ-Z1 designs.

It was previously assessed that the memory resources were the bottleneck when scaling the input size of the design. By applying cyclic array partitioning and decomposing the memory into BRAM and LUTRAM, the memory utilization was optimized. When utilizing nearly all the available BRAM and LUTRAM resources, at respectively 93 and 87 percent, in combination with the implemented memory optimization techniques, a design that can manage a resolution of 300 for 100 grid points was generated. Applying the pipeline optimization reduced the latency of the Forward and Cost function with, respectively, a factor of 9.37 and 2.95. The last step of the hardware design is to export the bitstream

and hardware handoff (HWH) file generated with Vivado. The HWH file contains information about the target platform and design that PYNQ requires to correctly configure the system and connect internal python methods to the corresponding features, signals, or drivers [49].

Optimization level	Resolution	Gridsize	LUT	FF	DSP	BRAM	LUTRAM
			(%)				
Initial design	**125**	**100**	**39**	**31**	**12**	**60**	**3**
Array partitioning (cyclic)	**200**	**125**	**26**	**16**	**14**	**89**	**17**
LUTRAM	**300**	**100**	**50**	**16**	**15**	**93**	**87**

Table 5.2: The resource and design specifications of the largest fitting design per optimization level for PYNQ-Z1. The FPGA resources are denoted as the percentage of total available resources.

The Alveo U280 FPGA's available resources are much higher than that of the PYNQ-Z1 board, as shown in Section 2.1.3. Table 5.3 shows the percentages of resources used for four Alveo U280 hardware designs with increasing resolution. Due to the higher available resources, it was possible to implement designs with up to 8000 resolution for a grid size of 100, while the PYNQ-Z1 only allowed for a resolution of 300. A complete overview of the utilization of all the FWI hardware designs made for Alveo U280 is provided in Table 6.2.

Resolution	Grid size	LUT	FF	DSP	BRAM	URAM	LUTRAM
1000	**100**	**0**	**0**	**0**	**0**	**0**	**0**
2500	**100**	**15**	**11**	**1**	**30**	**31**	**5**
5000	**100**	**15**	**11**	**1**	**41**	**42**	**5**
8000	**100**	**16**	**11**	**1**	**44**	**71**	**6**

Table 5.3: Resources for Alveo U280 implementation with resolutions 1000, 2500, 5000 and 8000 with 100 grid sizes.

As mentioned previously, the CNN bitstream was compiled with the FINN compiler for Alveo U280. In Table 6.3 an overview of the utilization of all the generated CNN hardware designs is given as a percentage of the total available resources for Alveo U280.

5.2. Hardware communication

This section describes how the kernels can be accessed from the host code through the hardware communication layer. The PYNQ approach for PYNQ-Z1 and Alveo U280 is first discussed, followed by the OpenCL approach for Alveo U280.

5.2.1. PYNQ

In this section, the PYNQ functions used to program the design and access the kernels are first discussed for the PYNQ-Z1 SoC FPGA and successively for the Alveo U280.

For PYNQ-Z1 FPGAs and other boards from the ZYNQ family, a PYNQ SD card image is required to get started. Pre-compiled images with the latest version of PYNQ, version 2.7 at the time of writing, are available on the PYNQ website [50]. A tutorial is also available to build a PYNQ image if there are custom requirements or a pre-compiled image isn't available for the targeted board. The pre-compiled ubuntu-based image comes with an installation of the python package `pynq` for Python 3.6. This package can be used in a regular python environment, or through a convenient IPython environment in Jupyter Notebook. At boot time, the image automatically programs a base overlay to the PL that consists of IP blocks to control the peripherals of the target board and connect them to the PS. To use PYNQ on FPGAs from the Alveo family, an image is not necessary. The requirements at the time of writing are a version of XRT higher than 2.3, an XRT-supported operating system with python 3.5.2 or

higher, as well as pip installed.

Accessing the hardware design is slightly different for both platforms. Listing 5.5 shows how to program the bitstream and execute the cost kernel on PYNQ-Z1 and Alveo U280. Lines 2 through 4 use the PYNQ `allocate` class to allocate memory to buffers in the host memory. The user needs to specify the shape and variable type for the buffer. It should be noted that the allocate function can take an extra argument specifying the memory bank on which the data should be allocated. Especially for the Alveo cards that offer HBM and DDR memory, this can be useful. Lines 7 and 8 show how the PYNQ `Overlay` class is used to download the bit file, of which the path is specified as ALVEO and PYNQZ1 in the listing, to the FPGA. In this example, the first available device found by the PYNQ `Device` class is targeted. The PYNQ-Z1 implementation requires the user to transfer allocated pynq buffers directly to the DMA channels designed in the kernels to start the computations. The DMAs are assigned in lines 11 through 13. The Alveo implementation only requires the user to specify the kernel, shown in line 16, and not the transfer channels as this is automatically handled when a buffer is synchronized to the device. Lines 19 through 28 show how the cost function kernel is launched for PYNQ-Z1. The buffers are transferred to the DMAs and wait until the computations are finished and transferred back to the main memory output buffer. In lines 31 through 37 the cost function is launched for the Alveo U280 implementation. As mentioned previously, the buffers are simply synchronized to the device and the kernel is called for execution. Once execution completes, the results are synchronized back to the host's main memory output buffer.

Listing 5.5: Example how to execute a kernel through PYNQ for PYNQ-Z1 and Alveo boards.

```
1  #Allocate memory for Cost function
2  residual_buffer = allocate(shape=(resolution,),dtype=np.complex64)
3  kappa_buffer = allocate(shape=(resolution,gridsize), dtype= np.
       complex64)
4  output_buffer = allocate(shape=(gridsize), dtype = np.complex64)
5
6  # Program the bitstream to the targeted device.
7  ol_alveo = Overlay(ALVEO, download=True, device=Device.devices[0])
8  ol_pynq = Overlay(PYNQZ1, download=True, device=Device.devices[0])
9
10 # Assign the DMAs for PYNQ-Z1
11 residual_dma = ol_pynq.residual_dma
12 kappa_dma = ol_pynq.kappa_dma
13 output_dma = ol_pynq.output_dma
14
15 # Assign the cost function for AlveoU280
16 cost_function = ol_alveo.cost_function
17
18 # Launch the cost function for PYNQ-Z1
19 def cost_function_PYNQ_Z1(kappa_buffer, residual_buffer,
       output_buffer):
20     # Link the allocated buffers to the DMA channels and start
           transferring
21     residual_dma.sendchannel.transfer(residual_buffer)
22     kappa_dma.sendchannel.transfer(kappa_buffer)
23     output_dma.recvchannel.transfer(output_buffer)
24
25     # Wait for the DMAs to finish transferring the results back to
           main memory
26     residual_dma.sendchannel.wait()
27     kappa_dma.sendchannel.wait()
28     output_dma.recvchannel.wait()
```

```
29
30 # Launch the cost function for Alveo U280
31 def cost_function_ALVEO_U280(kappa_buffer,residual_buffer,
       output_buffer):
32     # Synchronise the allocated buffer and launch the kernel
33     kappa_buffer.sync_to_device()
34     residual_buffer.sync_to_device()
35     ol_alveo.cost_function.call(kappa_buffer,residual_buffer,
           output_buffer)
36     # Synchronise the results back to main memory
37     output_buffer.sync_from_device()
```

5.2.2. OpenCL

The Vitis platform provides the user with the Xilinx OpenCL (XCL) extension to accelerate C and C++ applications with OpenCL. XCL is an extension that provides several mechanisms to interact with the FPGA and hardware design through XRT. In this section, the reader is guided through the implementation steps to accelerate the FWI hardware design using XCL.

As explained in Section 4.2, the hardware communication layer is used to download the hardware design to the FPGA and perform execution and memory management. XCL provides the user with utility functions to perform these tasks. In Listing 5.6 the commands to initialize the hardware communication are displayed. Error handling and device name checks have been omitted from this listing for the sake of clarity.

Listing 5.6: XCL and OpenCL initalization.

```
1     // returns the connected AMD platforms
2     auto devices = xcl::get_xil_devices();
3
4     // Select device [0]
5     device = devices[0];
6
7     // returns a pointer to the bitstream
8     auto fileBuf = xcl::read_binary_file(binaryFile);
9     cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
10
11    // Create the context and commandqueue
12    OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr,
          nullptr, &err));
13    OCL_CHECK(err, q = cl::CommandQueue(context, device, &err));
14
15    // Program the device in the context
16    cl::Program program(context, {device}, bins, nullptr, &err);
17
18    // Create the interface with the kernels in the hardware design
19    OCL_CHECK(err, krnl_cost= cl::Kernel(program, "Cost", &err));
20    OCL_CHECK(err, krnl_forward = cl::Kernel(program, "Forward", &err
          ));
```

OpenCL uses `contexts` to program, manage and interface with devices. Context consist of the targeted device(s) and `command queue` objects. Commands can be submitted to the latter to access the hardware kernels. The listing shows how XCL and OpenCL are used to find and select the targeted device, create the context and command queue, program the device, and acquire the kernels from the hardware design.

In Listing 5.7, the implementation to execute the cost function is shown. Memory is allocated for the

input and output vectors in the host and global memory. Filling the vectors with data has been omitted to keep the size of the listing to a minimum. To increase performance an aligned allocator is used. The data is copied to global memory and the cost function is launched. At completion the output is transferred back to host memory.

Listing 5.7: XCL and OpenCL implementation of the cost kernel execution.

```
1  // Allocate Memory in Host Memory
2  vector<complex_float,aligned_allocator<complex_float> > residual(
       ROW);
3  vector<complex_float,aligned_allocator<complex_float> > kappa(ROW
       *COL);
4  vector<complex_float,aligned_allocator<complex_float> >
       cost_output(COL,0);
5
6  // Allocate Buffer in Global Memory
7  // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient
       memory and Device-to-host communication
8  OCL_CHECK(err, cl::Buffer buffer_in_res(context,
       CL_MEM_USE_HOST_PTR,sizeof(complex_float)*ROW,
9                                          residual.data(), &err));
10 OCL_CHECK(err, cl::Buffer buffer_in_kappa(context,
       CL_MEM_USE_HOST_PTR, sizeof(complex_float)*ROW*COL,
11                                        kappa.data(), &err));
12 OCL_CHECK(err, cl::Buffer buffer_output(context,
       CL_MEM_USE_HOST_PTR , sizeof(complex_float)*COL,
13                                          cost_output.data(), &err))
                                               ;
14
15 OCL_CHECK(err, err = krnl_cost.setArg(0, buffer_in_res));
16 OCL_CHECK(err, err = krnl_cost.setArg(1, buffer_in_kappa));
17 OCL_CHECK(err, err = krnl_cost.setArg(2, buffer_output));
18
19 // Copy input data to device global memory
20 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_in_res,
       buffer_in_kappa}, 0 /* 0 means from host*/));
21
22 // Launch the Kernel
23 OCL_CHECK(err, err = q.enqueueTask(krnl_cost));
24
25 // Copy Result from Device Global Memory to Host Local Memory
26 OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_output},
       CL_MIGRATE_MEM_OBJECT_HOST));
27 q.finish();
```

The above-presented implementation was used to accelerate the C++ FWI implementation. The listing provided in this thesis was adapted for the sake of clarity. The original code base used in the experiments can be found at https://github.com/Luc-Dierick/FWI_Vitis.

5.2.3. Multiple compute units

The multiple compute unit feature from Vitis allows users to easily duplicate kernels if the FPGA resources allow it [27]. Under the hood, Vitis uses a Vivado-based compiler and linker called v++. By default, one instance of each kernel is created by v++. The `connectivity` setting of the v++ configuration file can be used to specify for each kernel how many instances should be created and to what memory banks the instance variables should be mapped. An example for two instances of the cost kernel connected to DDR0 and DDR1 is displayed in Listing 5.8.

Listing 5.8: Connectivity settings for two instances of the Cost kernel.

```
[connectivity]
nk=cost:2:cost_1.cost_2
sp=cost_1.kappa:DDR[0]
sp=cost_1.cur_est_pres_field:DDR[0]
sp=cost_1.output:DDR[0]

sp=cost_2.kappa:DDR[1]
sp=cost_2.cur_est_pres_field:DDR[1]
sp=cost_2.output:DDR[1]
```

To operate on multiple compute units, the hardware communication needs to be adapted. The following sections describe how this was done for OpenCL. In Section 5.4 this is explained for PYNQ in the context of OctoRay.

OpenCL command by default queues submit tasks in the order they were enqueued. To concurrently access different kernel instances on the targeted device, the command queue needs to be specified as an out-of-order queue. In FWI, there are many interactions between the host code and the hardware design. Multiple compute units give the possibility to execute sets of the cost and forward functions in parallel. To fully benefit from this, the host code needs to run in parallel as well. C++ native multiprocessing was used to set up as many host code processes as there are duplicated cost and forward functions.

5.3. Interface

To integrate with the pythonesque nature of OctoRay and PYNQ, a python implementation of FWI was necessary. To that end, the C++ implementation was rewritten in python. To keep the loss of performance to a minimum, NumPy libraries that use C-implementations under the hood were used as much as possible. For some functions in the standard C++ library, there exists no python equivalent. Two of these functions used in the FWI algorithm are the `cyl_neumann` and `cyl_bessel` functions [51] [52]. To incorporate these functions in the python implementation, a wrapper class was developed using `ctypes` and cross-compiled shared libraries of the functions. It should be noted that the python implementation only incorporates the implementation of the finite difference forward model and the conjugate gradient descent inversion model.

5.4. OctoRay

Previous to this work, the OctoRay project was a collection of Jupyter Notebooks that demonstrate how to use Dask distributed in combination with PYNQ to parallelize data analytics applications on FPGAs using open-source frameworks. While the concept of OctoRay is an excellent initiative to make FPGA acceleration scalable and accessible, the initial framework served as a proof of concept and there remained a lot of room for improvements. This section begins by discussing the original architecture of OctoRay and its advantages. As the improved version of the framework has a different setup and installation, the steps to replicate the original OctoRay are only discussed at a high level. The current installation and environment setup are discussed in Section 5.4.3. The original framework was developed by Shashank Aggarwal, his work can be found on the TU Delft repository [53]. OctoRay exploits the concept of data parallelism. Figure 5.5 shows the general architecture applied for two workers.

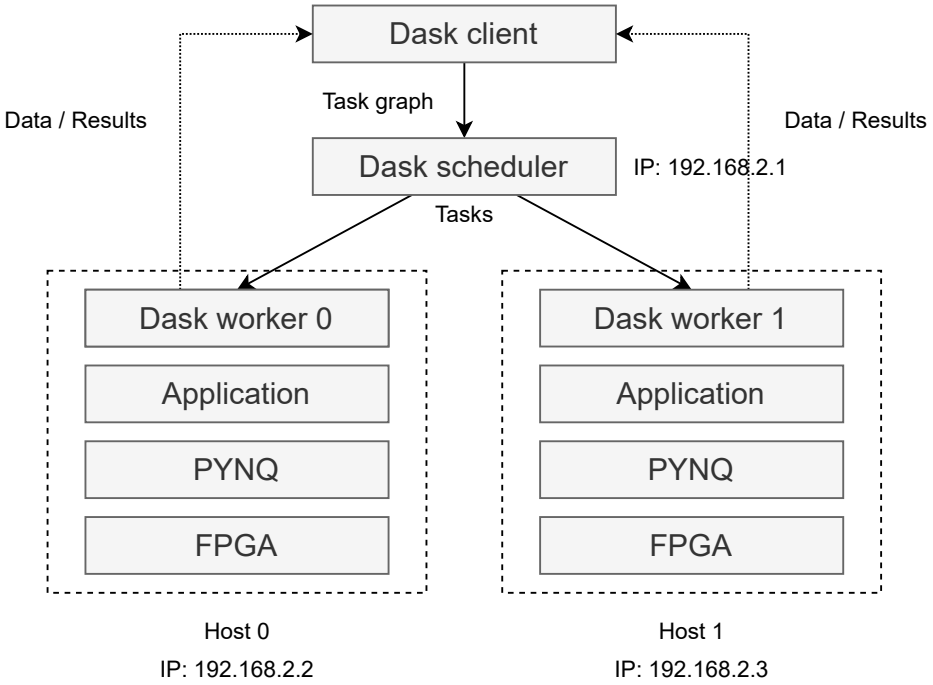

Figure 5.5: The general architecture of the OctoRay framework, figure ada[ted from [53]

The architecture consists of three key components, the dask client, scheduler, and workers. It should be noted that OctoRay relies on the distributed dask library, to prevent redundancy from now on when dask terminology is used the distributed versions are meant unless specified otherwise. To prevent misinterpretations between the code related to the parallelization framework and to the accelerated application, the former is referred to as OctoRay host code and the latter as the python driver. A python driver consists of the hardware design, called an overlay in the PYNQ context, and the application host code. The host code incorporates the hardware communication layer to interact with the overlay. It is assumed at this stage that the bitstreams containing the hardware design and python, PYNQ, Dask, and other requirements are present on the FPGA-enabled nodes.

In the following steps, a high-level overview of how the original OctoRay framework was used to scale an FPGA application on multiple nodes.

1. Instantiate a cluster by spawning one scheduler, and as many workers as necessary. To perform this step, the user must have access to a terminal on each node to manually start the respective dask processes.
2. Create a client in the OctoRay host code that serves as the primary entry point to the cluster.
3. The client automatically detects the number of workers available in the cluster. The input data is divided by the number of workers and scattered among them. For each worker, the client sends the pynq driver to the scheduler as a task. The scheduler distributes and schedules the tasks for execution on the workers.
4. When a worker completes its task, the output is returned to the client. Once the results from all workers have been received, they are merged together and made available to the user in the OctoRay host code.

OctoRay was proven a working concept through two functional examples of scaled data analytics applications available at https://github.com/abs-tudelft/octoray. Several areas of improvement were identified. In the following sections, they are discussed and the associated implemented solutions are presented.

OctoRay was built with the purpose of scaling a single application on multiple of the same FPGAs. Although supporting the deployment of different types of FPGAs, it does not allow the combination of

different applications and FPGAs in a heterogeneous infrastructure. With the lack of a general structure, users require a deeper knowledge of OctoRay components such as Dask cluster setup and data and task management.

Two main components were introduced to improve the deployment and usability of the framework, respectively, a cluster configuration and OctoRay kernels. Together, these components form the foundation of the new architecture. First, the changes in the deployment of OctoRay by using the cluster configuration are discussed. Next, the flexibility and increased usability enabled by the OctoRay kernels are presented.

5.4.1. Deployment

Dask provides the user with multiple ways to set up a distributed cluster presented below.

1. Manual setup of dask processes through CLI.
2. Managed services such as Kubernetes and Hadoop Yarn.
3. Dask SSH cluster API.
4. Docker images

The original framework chose the manual setup of dask processes. For a small number of hosts, it was an easy, fast but most importantly flexible manner of deployment that enabled access to low-level functionality of the dask processes. Setting up managed services or using docker images requires extra user effort which is exactly what OctoRay tries to minimize. A drawback of the manual setup and tear-down of dask processes is that it inhibits large-scale deployment. For a small amount of workers manual instantiating and tearing down workers is a feasible task, but as soon as the number of workers exceeds double digits this becomes extremely tedious and time-consuming.

The Dask SSH cluster API is an experimental project from Dask to deploy dask clusters from python over SSH connections [54]. A custom implementation of the Dask SSH Cluster was made to fit OctoRay's needs. OctoRay lets the user specify the deployment with a cluster configuration dictionary, displayed in Listing 5.9.

Listing 5.9: Example cluster configuration for two hosts each containing one worker.

```
1 cluster_config = {
2     "scheduler":"10.1.212.127",
3     "hosts":["10.1.212.126","10.1.212.127"],
4     "connect_options":{"port":22,"xrt":XRT_ENV_PATH,"dir":SPAWN_PATH
          },
5     "worker_options":{"nthreads":1,"n_workers":1,"preload":"
          pynqimport.py","nanny":"0","memory_limit":0},
6     "scheduler_options":{"port":8786},
7     "worker_class":"distributed.Worker",
8     "overlay": Bitstream.xclbin
9 }
```

The cluster configuration is multi-functional. Primarily, it allows users to specify the IP addresses of the machines where the dask scheduler and workers should be deployed. For automatic deployment of the cluster, several settings must be specified, dictated by the Dask SSH Cluster implementation [55]. In this example, a dictionary is passed as connect options indicating that these are the same for all nodes, called hosts, in the cluster. In the case of different connection options, a list can be provided with custom settings. In the case of a list, a dictionary of settings must be specified for each host. The worker options behave in the same way and are used to specify the behavior of workers. The Dask SSH Cluster was extended with two features. First, the connect options were expanded with the `xrt` and `dir` options to specify where the XRT environment should be sourced from and in what directory the

worker processes should be spawned. Dask workers natively perform their work in a non-main thread. PYNQ requires to be imported from the main thread to function correctly. The old OctoRay solved this problem by spawning a new process in the python driver where the application code using PYNQ was executed. To circumvent this user-burdening solution, a preload script is passed to the workers in which PYNQ is imported and the specified Overlay is downloaded to the FPGA. This feature increases usability by removing the effort to create a process for the user and automating the download of the bitstream. Another advantage of this method is discussed in the following section when the reader is led through an example usage of OctoRay.

5.4.2. Flexibility and usability

The second core component of the new OctoRay architecture is the OctoRay kernel. The cluster configuration is meant to facilitate the general deployment of the cluster. OctoRay kernels are structures that consist of all the specifications necessary to deploy an application on the cluster. Listing 5.10 shows the function definition to create an OctoRay kernel.

Listing 5.10: Function definition to create an OctoRay kernel

```
1  def create_kernel(self, path:str, no_instances:int=1, batch_size:int
       =0, func_specs:list=[],config=None,device:str=None):
2          """Creates a dictionary that represents a kernel.
3          @param device: The device name of the FPGA
4          @param path_to_bitstream: The path to the bitstream
5          @param no_instances: If there are copied instances (default =
               1)
6          @param instance_id: The id of the compute unit, this is 1 or
              configured based on no_instances.
7          @param batch_size: The amount of data each compute unit
              should process.
8          @param func_specs: The functions inside the kernel with their
               memory specifications
9              Example: A function square_numbers(double a, double b)
                  where a is mapped to HBM0 and b to HBM1
10             is represented as: [{"square_numbers":[HBM0,HBM1]}]
11         @param host: We assign each kernel to a host, chronologically
               by default.
12         @param config: If the python driver requires it, a
              configuration file or variable can be added.
13         """
14         kernel = {
15             "device":device,
16             "path_to_bitstream":path,
17             "no_instances":no_instances,
18             "instance_id":1,
19             "batch_size":batch_size,
20             "functions":func_specs,
21             "host":None,
22             }
23         if config:
24             kernel["config"]=config
25
26         return kernel
```

OctoRay kernels were developed in combination with helper functions to increase the usability and flexibility of the architecture. The kernel specifications can be read in the function description in Listing 5.10. An import feature of the kernels is that they transparently support hardware designs containing

multiple copied instances. A user can specify the number of copied instances a hardware design contains, and the batch size per compute unit. OctoRay takes care of the rest of the configuration and deployment. The details of this process are discussed when the helper functions are introduced in the following paragraphs.

The framework sees an OctoRay kernel as an entity that can be executed on a host. An important advantage of grouping all the application and execution specifications in a kernel is that this enables users to define multiple configurations of applications in the same context. In the previous architecture, the functions and specifications were directed at one specific application for all hosts in the cluster, resulting in a rigid framework. With OctoRay kernels, a user can specify for each host what application and associated bitstream and dataset should be executed.

Listing 5.11 shows how OctoRay can be used to execute two different configurations of the CNN application. In line 5, the OctoRay object is created with arguments specifying the automatic cluster deployment and the cluster configuration dictionary. Line 8 is where the actual cluster is deployed in the case of automatic deployment, if the cluster is already deployed this function creates a dask client to connect to the existing cluster. In line 11 the cifar10 data is set is loaded, which is split in batch specified by the `BATCH_SIZE` macro, the actual numbers are irrelevant to this example. In line 14 a list of CNN kernels is created, in this example, the deployment of one type of application with different configurations is demonstrated. If the user wishes, OctoRay kernels for different applications and configurations can be added to other lists. Line 17 through 19 demonstrate how OctoRay kernels of the same configuration are created and appended to the CNN kernels list. Lines 20 through 23 demonstrate how two differently configured kernels are added. In line 25 the worker options are specified for each host based on the connect and worker options specified in the cluster configuration.

The function in line 28 subdivides kernels containing hardware designs with multiple compute units into individual kernels per compute unit. Additionally, the worker options for the associated hosts are reconfigured to match the independent compute unit kernels. This ensures that a dask worker is spawned for each compute unit on the correct host. Note that in the case of automatic deployment, the `n_workers` setting of the worker options for a host associated with a kernel containing multiple compute units needs to align with the number of compute units. Dask workers are independent software processes that operate in parallel. The processes need to acquire access to the hardware design to accelerate an application. For a single instance hardware design, associated with one worker, this is a trivial task. For designs with multiple compute units, multiple software processes need to access the hardware design in parallel. PYNQ uses the Overlay class to program a device or to access an already programmed device. If two processes try to program a device at the same time, a race condition occurs and PYNQ crashes in both processes. To circumvent this issue, the bitstream is downloaded to the target device during cluster setup by using a preload script. Dask sequentially instantiates its workers during deployment, which means no race conditions occur. The first worker actually programs the bitstream and the following workers retrieve the bitstream's context. As mentioned before, this also results in the advantage of PYNQ and the bitstream context being present in the main thread of the workers.

In line 34 the data set is transparently split into batches that match the requirements of each kernel according to its configuration. The user does not need to differentiate a multiple compute unit design from a single instance design as OctoRay has configured this. In line 37 the data and associated kernels are scattered to the workers. The `run_on_worker` function needs to be defined and consists of the python driver to execute the application. In line 37 the retrieved results from the workers are sorted to match the original input order.

Listing 5.11: Example use of OctoRay kernels to accelerate two identical or different configurations of CNN

```
1
2 from octoray import Octoray
3
4 # Initialiaze OctoRay with automatic cluster deployment
```

```
5  octoray = Octoray(ssh_cluster=True, cluster_config=cluster_config)
6
7  # set up the cluster
8  octoray.create_cluster()
9
10 # load dataset
11 full_cifar = np.load('cifar10.1_v4_data.npy')
12
13 for BATCH_SIZE in BATCH_SIZES:
14     cnn_kernels = []
15
16     # Both hosts execute a kernel with a single instance of the
       hardware design
17     for i in range(len(cluster_config["hosts"])):
18         cnn_kernels.append(octoray.create_kernel(XCLBIN_PATH_SINGLE
           ,1,int(BATCH_SIZE/octoray.num_of_workers),[[{"idma0":["
           HBM0"]},{"odma0":["HBM0"]}]], device=DEVICE_NAME_DEFAULT))
19
20     # The first host executes a single instance design, the second a
       double instance design
21     cnn_kernels.append(octoray.create_kernel(XCLBIN_PATH_SINGLE,1,int
       (BATCH_SIZE/octoray.num_of_workers),[[{"idma0":["HBM0"]},{"
       odma0":["HBM0"]}]], device=DEVICE_NAME_DEFAULT))
22     cnn_kernels.append(octoray.create_kernel(XCLBIN_PATH_DOUBLE,2,int
       (BATCH_SIZE/(octoray.num_of_workers)), [[{"idma0":["HBM0"]},{"
       odma0":["HBM0"]}],[{"idma1":["HBM1"]},{"odma1":["HBM1"]}]],
       device=DEVICE_NAME_DEFAULT))
23
24     # Set up the host's worker options configured in the cluster
       configuration file.
25     octoray.setup_worker_options()
26
27     # In the case of multiple instances, subdivide the kernels over
       the number of instances and update the worker options.
28     kernels_split = octoray.split_kernels(cnn_kernels)
29
30     # Divide the data set over the kernels based on batch size per
       instance
31     data_split = octoray.split_data(full_cifar[:BATCH_SIZE],
       kernels_split)
32
33     # Launch the tasks after scattering the data and kernels to the
       correct workers
34     results = octoray.execute_hybrid(run_on_worker,data_split,
       kernels_split)
35
36     # Reorder the response based on the original input order
37     results.sort(key = lambda result: result['index'])
```

5.4.3. Accessibility

The OctoRay project was developed with the goal to be an accessible, open-source framework aiding in the adoption of FPGAs. To further increase the accessibility and ease of setup, the collection of jupyter notebook examples was turned into an installable package from the Python Package Index (PyPI). The library implements constructs and functions that improve the flexibility, accessibility, and deployment of the framework as described in the previous sections.

6

Results

This chapter discusses the results from experiments performed on the different solution architectures presented in Chapter 4 for the applications FWI and CNN presented in Chapter 3. Figure 6.1 displays the topology of hardware platforms and applications that are compared and discussed in this chapter. The chapter begins with a description of the experimental setups for the different applications in Sections 6.1 and 6.2. Successively, the results for FWI are presented in Section 6.3 and for CNN in Section 6.4.

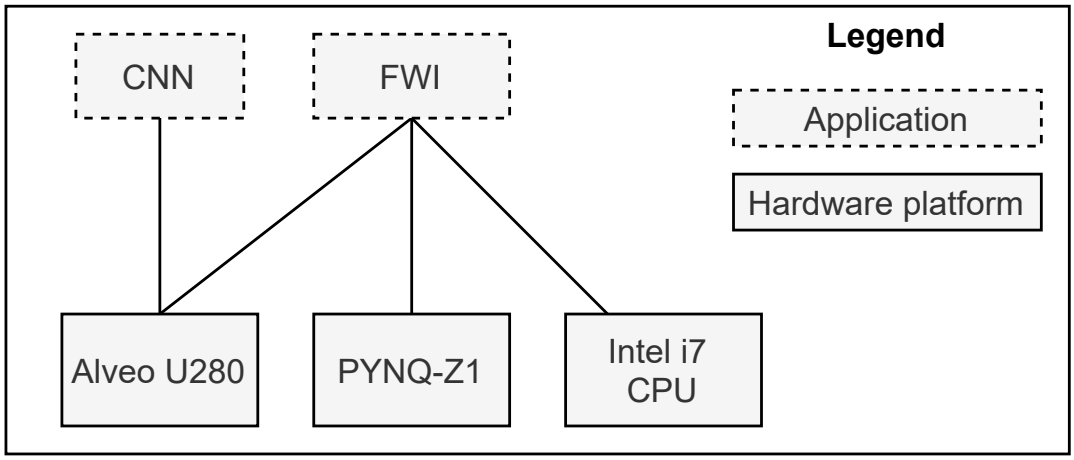

Figure 6.1: Topology of hardware platforms and applications.

6.1. Experimental setups for FWI

In this section, the experimental setups for the different hardware platforms are presented. In Section 6.1.1, the laptop and CPU used for the baseline of FWI are presented. In Section 6.1.2, the experimental setup of multiple PYNQ-Z1 boards exploited by OctoRay is described. Subsequently, the setup used to measure the power consumed by a PYNQ-Z1 while executing FWI is introduced. Lastly, Section 6.1.4 describes the setups used to scale and compare FWI for different configurations on multiple Alveo U280 FPGAs.

6.1.1. CPU

In order to evaluate the hardware-accelerated FWI implementation, experiments with the python and C++ implementations were performed on a high-end CPU to form a baseline. An HP Z-Book studio G6 laptop with an Intel I7-8750h CPU containing 12 logical cores running at 2.20 GHz with 16GB of RAM was used for this purpose. The C++ implementation used for the CPU is not available publicly. Permission was granted for the FPGA and OpenCL adapted C++ implementation and the python implementa-

tion. They are available at, respectively, https://github.com/Luc-Dierick/FWI_Vitis and https://github.com/Luc-Dierick/FWI-python.

6.1.2. PYNQ-Z1 OctoRay

The PYNQ-Z1 OctoRay setup consisted of a local network set up with 4 PYNQ-Z1 boards connected through category 6 (10Gbps) ethernet cables to a Sweex SW105 5 port Gigabit switch. The host PC, an HP Zbook Studio G5 containing the previously mentioned Intel I7 CPU was also connected to the switch. Internet access was shared through the host PC to the boards. On each board, the bitstreams and the FWI-PYNQ repository were downloaded, containing the FWI-python repository, jupyter notebooks for single machine visualization, and a pynq driver. The old OctoRay architecture was used for these experiments. The pynq driver served as the entry point for dask workers. The driver is responsible for downloading the bitstream to the FPGA, setting up the DMAs, and initializing and executing the python implementation of FWI. The driver returns metrics about the run and the output of the algorithm, a reconstructed subsurface grid velocity model.

6.1.3. PYNQ-Z1 power measurement

To investigate the power consumption of the FWI hardware design, a power-measurement setup was used. The setup was created by the Digital Technology Group from the Department of Electrical Engineering and Computer Science of the University of Kassel, Germany. The explanation and figures covering the setup were made available by the group, but are not yet publicly available. An overview of the setup is shown in Figure 6.2.

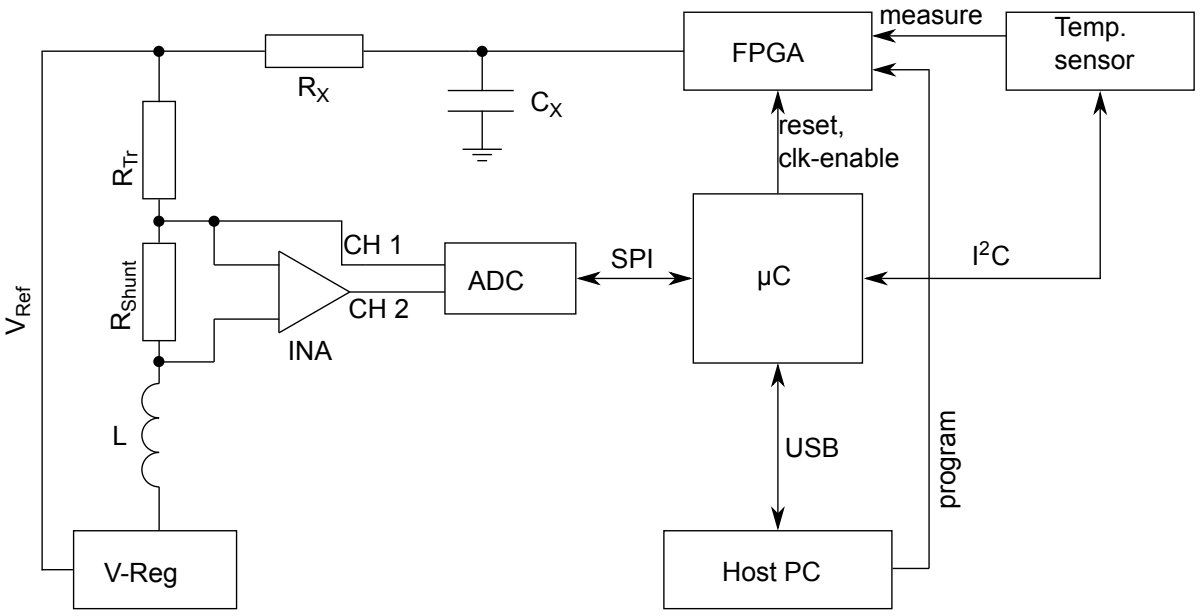

Figure 6.2: An overview of the power measurement setup for PYNQ-Z1 from the DTG group of University of Kassel.

The setup measures the dissipated power from the FPGA by using a precise shunt resistor (R_{shunt}) for the current measurement, followed by a potential measurement after the resistor. The voltage regulator (V-Reg) measures the voltage in the 1V domain and puts additional voltage to compensate for the voltage drop over the resistor and the inductance. This is done to inhibit the shunt resistor from having a negative effect on the 1V-domain. A highly precise and calibrated instrumentation amplifier (INA) is used to reduce the offset effects of measurement devices. Both the amplified voltage across the shunt and the potential after the shunt, which is the core voltage of the FPGA, are digitized with the 24-Bit-ADC ADS131M02 from TI. The maximal possible sampling rate is 32 kHz. The micro-controller (μC) ESP8266-D1-Mini acquires the data via SPI. In addition to that, the micro-controller collects temperature measurements from a connected infrared sensor MLX90615 from Melexis for temperature monitoring via an I2C-Bus. Finally, the measured values are made available to the Host PC via a USB connection. It should be noted that only the 1 Volt domain, which supplies the PL of power is mea-

sured. Other domains that power for example the DDR3 memory can not be measured in the current setup. The DTG group is working on an extended setup that measures all the voltage domains.

The experiments performed on this setup were used to investigate the power consumption of FWI on PYNQ-Z1. The experiments were performed for two hardware designs with differing resource utilization. displayed in Table 6.1.

Design	Resolution	Grid size	LUT	LUTRAM	FF	BRAM	DSP
High utilization	**300**	**100**	**50**	**87**	**16**	**93**	**15**
Low utilization	**300**	**20**	**28**	**23**	**15**	**31**	**12**

Table 6.1: An overview of the two designs and their utilization used for the power measurement experiments. The resources represent the percentage of available resources used.

As established in Chapter 5, changes in resolution have a much larger effect on the number of iterations necessary for FWI to converge. Keeping the number of iterations as constant as possible is important for a fair comparison of the two designs. Therefore, the resource utilization was reduced by decreasing the number of grid points processed and leaving the resolution intact. Additionally, it should be noted that although the designs allow for a resolution of 300, in practice a resolution of 294 made up of 7 sources and receivers and 6 frequencies were used. The implementation requires the numbers of sources and receivers to be equal which means a resolution of 300 could be built by using 10 sources and receivers and measuring over 3 frequencies. Measuring over that few frequencies leads to unreliable results which is why it was chosen to use a resolution of 294. To further improve the reliability of the results, all the experiments performed on this setup were replicated 5 times, and the presented numbers are averaged over the runs.

6.1.4. Alveo U280

The Alveo experiments were performed on the HACC cluster at ETH University, Zurich [20]. The cluster consists of multiple servers that give access to different types of FPGAs. For this work, up to four Alveo U280 FPGAs were used that are connected by an intra-cluster high-speed network implemented by two 100 Gbps connections to each Alveo card. In the background chapter in Section 2.1.3 an overview was given of the server-grade host processor and Alveo U280 total resources.

The OpenCL experiments were performed on a single Alveo U280 to assess the performance of the C++ implementation used in combination with the OpenCL hardware communication layer. The Alveo U280 OctoRay experiments were also performed on the HACC cluster, utilizing up to 4 Alveo U280s. In Table 6.2 an overview is given of all the different hardware designs for Alveo U280 used in this work. The overview displays the percentage of resources used for different configurations that are defined by the resolution, grid size, and the number of compute units.

Resolution	Grid size	CUs	LUT	LUTRAM	FF	BRAM	URAM	DSP
8000	100	1	16	6	11	44	71	1
5000	**100**	**1**	**15**	**5**	**11**	**41**	**42**	**1**
5000	100	2	18	7	13	73	82	1
2500	**100**	**1**	**15**	**5**	**11**	**30**	**31**	**1**
1000	100	1	15	5	12	23	21	1
1000	**100**	**2**	**17**	**5**	**13**	**31**	**44**	**1**
1000	200	1	15	5	12	31	29	1
300	**100**	**1**	**14**	**4**	**11**	**17**	**4**	**1**

Table 6.2: An overview of the percentage of total utilization for different Alveo U280 FWI hardware designs generated with Vitis.

6.2. Experimental setup CNN

The CNN was later introduced in this Master's thesis, with the purpose to showcase the applicability of OctoRay on a single-input dependent application. Therefore, the experiments performed with CNN were only performed on up to 4 high-end Alveo U280 hardware platforms on the previously mentioned HACC cluster. The experiments performed for the CNN applications were all performed 5 times and the average results are presented.

In Table 6.3 an overview is provided of the utilization of all the CNN hardware designs used in the experiments.

# of CNN instances	LUT	FF	DSP	BRAM	LUTRAM
1	10	7	<1	15	2
2	12	9	<1	19	3
4	21	17	<1	36	6
8	30	24	<1	58	8
10	35	27	<1	71	9

Table 6.3: An overview of the percentage of total utilization for CNN hardware designs containing different numbers of compute units.

6.3. FWI results

Experiments were performed for different configurations of the FWI application on four hardware platforms. First, in Section 6.3.1 an analysis is provided of how data parallelism can be applied to FWI using domain decomposition and what the effects are on performance and result quality. Having established that FWI can be scaled using domain decomposition, Section 6.3.2 provides an overview of the effects of horizontal and vertical scaling on low and high-end FPGAs. Additionally, a distributed composition of PYNQ-Z1 low-end FPGAs is compared to a single high-end Alveo U280 FPGA in terms of performance. Subsequently, in Section 6.3.3, an implementation of FWI that utilizes most of the resources available on a high-end Alveo U280 FPGA is compared to the CPU baselines. In Section 6.3.4, an analysis is provided of the cost and utilization for the distributed PYNQ-Z1 composition, Alveo U280, and a personal laptop. Next, Section 6.3.5 presents the design-build duration of FWI for low and high-end FPGAs. Lastly, in Section 6.3.6 the power and energy consumption of the targeted hardware platforms are evaluated and discussed for different FWI configurations.

6.3.1. Domain decomposition

In Chapter 3, FWI was characterized as a multi-input-dependent application. It was established that by decomposing the input grid into local domains, data parallelism could be applied. In this section, analysis is performed on the effects of data parallelism on FWI in terms of performance and quality of the results.

For this analysis, FWI was applied to a synthesized input of a Delphi temple displayed as the left plot in Figure 6.3, an example of synthetic data originating from [32]. The temple is located in a 64 by 32 subsurface grid, where 64 is the width (x) axis and 32 is the depth (z) axis. Each grid cell represents $5m * 5m = 25m^2$, resulting in a total area of $64 * 32 * 25m^2 = 51200m^2$. The background consists of a homogeneous medium with a constant acoustic velocity of $c_0 = 2000m/s$. The temple, also a homogeneous medium, has a constant acoustic velocity of $c = 2218m/s$. To express the difference in velocity a contrast function χ is used that was introduced in Equation 3.1. From this equation, the temple's velocity in contrast to the background is $\chi \approx 0.187$. Figures 6.3, 6.4 and 6.5 show three different configurations of FWI applied to the Delphi temple data set. The left plot in each of these figures represents the real subsurface grid's velocity properties. The middle plot is reconstructed by using FWI on pressure field data that was synthetically generated from the original temple data set. The right plot in these figures displays the differences between the real and reconstructed subsurface grid expressed by the contrast function.

In Figure 6.3 the temple was reconstructed by applying FWI with a resolution of 6000 made up of 20 sources, 20 receivers, and 15 frequencies. In this configuration, a full inversion was performed without

applying domain decomposition using the C++ implementation of FWI. This configuration will be used as a baseline to compare the effects of lower resolutions and domain decomposition. It should be noted that domain decomposition neglects the non-linear properties of the model which normally affects the results in a negative way. The synthetic data sets with homogeneous velocities used in this work are far from realistic. The purpose of this work is not to optimize or improve the results of FWI but rather to show the possibilities of scalability, for which the synthetically generated data sets are well suited.

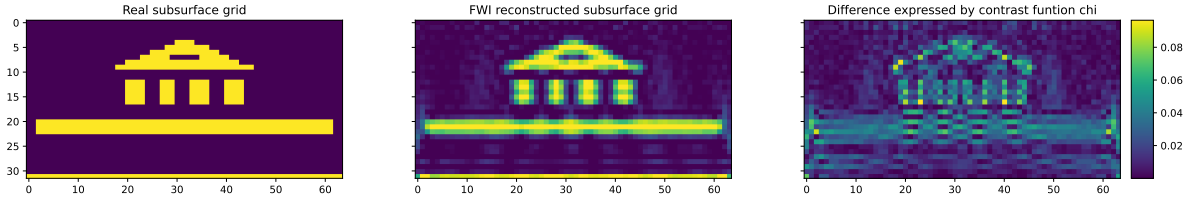

Figure 6.3: The real subsurface grid of the Delphi temple (left), the reconstructed grid (center), and the difference between the real and reconstructed grid (right) are expressed in chi. The C++ FWI implementation was applied on the full data set of grid size $64 * 32 = 2048$ with a resolution of sources, receivers, and frequencies $20 * 20 * 15 = 6000$.

In Figure 6.4, the same configuration as Figure 6.3 was used except that a drastically lower resolution of 294 was used, consisting of 7 sources and receivers and 6 frequencies. From the reconstruction and difference plots, it can be seen that the quality of the result for a resolution of 294 is much lower than its 6000 resolution counterpart. The pillars under the temple roof are the most difficult area to reconstruct as they are partially hidden and their orthogonal positions propagate reflected waves that result in complex non-linear properties. From these two figures, it becomes clear that the resolution needs to be of a high enough degree to be able to make sense of the reconstructed grid.

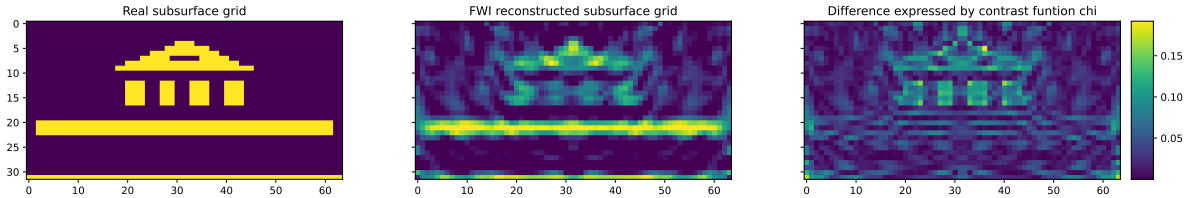

Figure 6.4: The real subsurface grid of the Delphi temple (left), the reconstructed grid (center), and the difference between the real and reconstructed grid (right) are expressed in chi. The C++ FWI implementation was applied on the full data set of grid size $64 * 32 = 2048$ with a resolution of sources, receivers, and frequencies $7 * 7 * 6 = 294$.

In Figure 6.5, FWI was applied to the sample temple data set and like Figure 6.4 a resolution of 294 was used. The core difference is that in Figure 6.5 domain decomposition was applied. The input data set was sliced into four layers, which are depicted by white horizontal lines in the figure. Each of the four-layer is 8 grid cells deep and 64 wide, together covering the total area. FWI is applied to each of these layers. By using a method called back-propagation, each layer can be independently inverted irrelevant of its location in the original subsurface grid. The inverted layers were then merged back together and displayed in the figure. The quality of the reconstructed grid has significantly increased by applying domain decomposition as can be seen when comparing the center and right plots of Figures 6.4 and 6.5 or in Figure 6.6. As mentioned previously, for this synthetically generated data set, the enhanced quality from domain decomposition should be taken with a grain of salt. The chi difference plot in Figure 6.3 shows that for a full subsurface inversion with a high resolution, there is a minimal amount of noise. Contrarily, the 300 resolution full and decomposed configurations show a high degree of noise in random places around the grid. It can also be observed that the edges of the temple are not as defined for the full inverted configurations as for the decomposed model. This occurs because with the decomposed model a smaller grid size is inverted that is only affected by the non-linear properties of that local grid, and not of the entire grid. The non-linear properties add to the blurriness around the edges but do reduce the noise in locations further away from the edges.

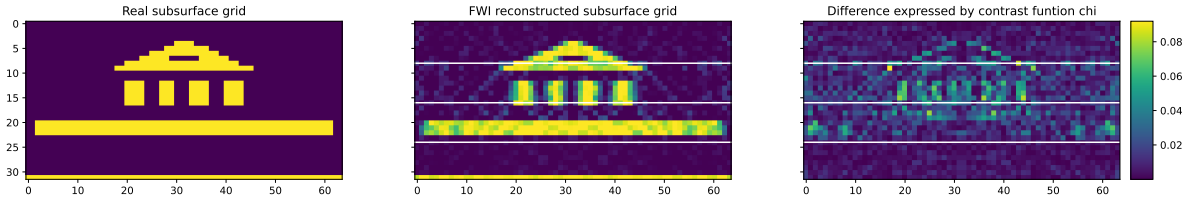

Figure 6.5: The real subsurface grid of the Delphi temple (left), the reconstructed grid of four independently inverted layers merged back together (center) and the difference between the real and reconstructed grid (right) is expressed in chi. The C++ FWI implementation was applied four times on $64 * 8 = 512$ grid size layers of the total data set. Each layer was inverted with a resolution of sources, receivers and frequencies $7 * 7 * 6 = 294$ and are separated by white lines in the figure.

The three configurations presented above were produced with the C++ implementation of FWI. The python implementation was used to reproduce these experiments and compare the results with the C++ implementation in terms of accuracy. The left plot in Figure 6.6 displays the average difference in chi for the three configurations executed in python and C++. As average numbers are taken for the entire subsurface grid, these results do not differentiate whether the noise is in random places or at the edges but they do give a general indication of the quality of the reconstruction. The right plot displays the execution time for the different configurations for the C++ implementation. The python execution times are not incorporated in this plot as its purpose is to show the difference in execution times for different FWI configurations. In the following sections, an elaborate analysis of the different FWI implementations and hardware platforms will be presented.

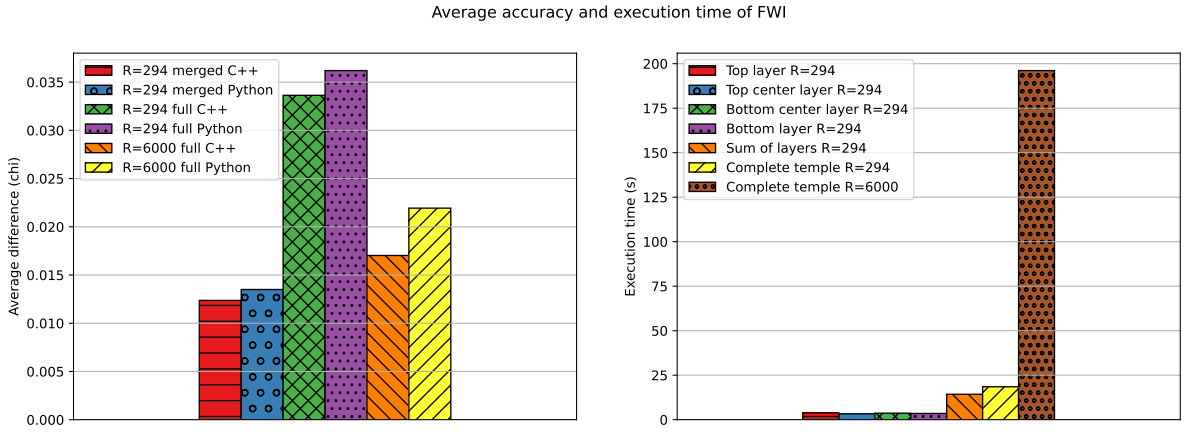

Figure 6.6: The accuracy of the CPU implementations is expressed through the contrast function chi (left). The execution time of the C++ implementation for the independent and summed layers of the merged model with a resolution (R) of 294 and of the complete data set with resolutions R=294 and R=6000 (right).

The right plot of Figure 6.6 shows the execution time independently and summed for the four layers of the merged model as well as the full models with resolutions of 300 and 6000. When considering the accuracy of the full model inversions, the left figure shows that increasing the resolution from 294 to 6000 improves the average accuracy by about 50%. This improved accuracy comes at a cost displayed in the right figure, where the execution time of the 6000 resolution full model inversion is approximately 10 times higher than of the 294 resolution model. From the left plot, it can also be read that the average accuracy of the merged model is the best overall. For all models, the python implementation results are of slightly worse accuracy because the computations are performed on `floats` with 32-bit precision instead of `doubles` with 64-bit precision in the C++ implementation. The python implementation was implemented with `floats` because the HLS tools used to implement the hardware designs did not support complex `doubles` used to represent the pressure field data. As the C++ implementation was already provided by ALTEN it was not altered. The right plot shows that the execution time for the independent layers summed together is also less than that of the full model with the same resolution of 294. Additionally, the independent layers can be processed in parallel reducing the execution time even further.

In summary, the following conclusions were derived from the experiments in this section:

- Domain decomposition can be used to divide the total subsurface grid into independently invertable local grids that can be processed in parallel.
- Decreasing or increasing the resolution leads to, respectively, lower and higher accuracy.
- Decomposition neglects the non-linear properties over the total subsurface grid leading to more dispersed noise. Because the non-linear properties of the locally inverted grid are taken into account, the edges are better defined.
- Increasing the resolution results in a higher complexity and longer execution time.

6.3.2. Scalability

In this section, the scalability of FWI is presented. It was established that domain decomposition can be used to divide the FWI problem size into batches that can be processed in parallel. There are three ways in which domain decomposition can be used to scale FWI. Primarily, the decomposed domains can be processed sequentially on one node. Next, the decomposed domains can be processed in parallel on multiple nodes, which is called horizontal scaling. Lastly, the decomposed domains can be processed in parallel on one node by using multiple instances of FWI in one hardware design which is called vertical scaling. This section begins by analyzing the effect of horizontal scaling on multiple PYNQ-Z1 nodes compared to the software baselines and a single Alveo U280 instance. Successively, the effect of vertical scaling is demonstrated by comparing it with horizontal and sequential scaling on Alveo U280 FPGAs.

Horizontal scaling

In Chapter 5, the hardware design for FWI was introduced. An analysis of the computational load of different functions led to a design consisting of two kernels for the forward and cost functions. From Figure 5.1, it was observed that the degree of resolution has the highest effect on the time spent in the forward and cost functions relative to the total run time. Because the algorithm is only partially accelerated in hardware, the processor that executes the host code also contributes to the total performance. Consequently, the host processor's contribution to the total performance is correlated with the percentage of the total computational load performed in the accelerated functions. In Figure 6.7, a similar analysis for the computational load is presented for a configuration with resolution 294 and grid size 100 for CPU as well as PYNQ-Z1 and Alveo U280 SoC FPGAs. The figure shows the time spent in the hardware-accelerated functions and the total execution time for increasing grid sizes. The CPU plot can be considered as a baseline because the hardware functions and the host part of the algorithm are performed on the same hardware platform. To make sure the comparison is reliable, the python implementation was used for the CPU, and in combination with the pynq hardware communication layer for the FPGAs. For the PYNQ-Z1 the host part is executed on a dual-core ARM Cortex A9 processor and the Alveo U280 on an Intel Xeon Gold server-grade processor. For this FWI configuration, the figure shows that the time spent in the hardware functions for the PYNQ-Z1, CPU, and Alveo U280 is approximately constant at 9%, 68%, and 80% respectively, when increasing the grid size. It should be noted that increasing the grid size in this context means iterating sequentially over 100 grid size batches. As the total execution time is linear when processing an increasing amount of 100 grid size batches, the throughput for this configuration can be computed for PYNQ-Z1, CPU, and Alveo U280 at respectively 21, 245, and 658 grid cells per second. It should be noted that these numbers are based on a warm start which means the overhead of programming the bitstream to the FPGA and initializing the algorithm is left out.

The FPGA resources of the PYNQ-Z1 form a restricting factor for the resolution and grid size of FWI configurations. It was shown that domain decomposition can be used to circumvent the grid size restriction. From Figure 6.7, the throughput in grid cells per second was established for the three hardware platforms. In Figure 6.8, the left plot displays the execution times of different numbers of PYNQ-Z1s processing a total subsurface grid in parallel by using OctoRay. The increasing grid sizes are once again formed by iterating over 100 grid size batches but also divided over 1, 2, or 4 nodes. The left plot

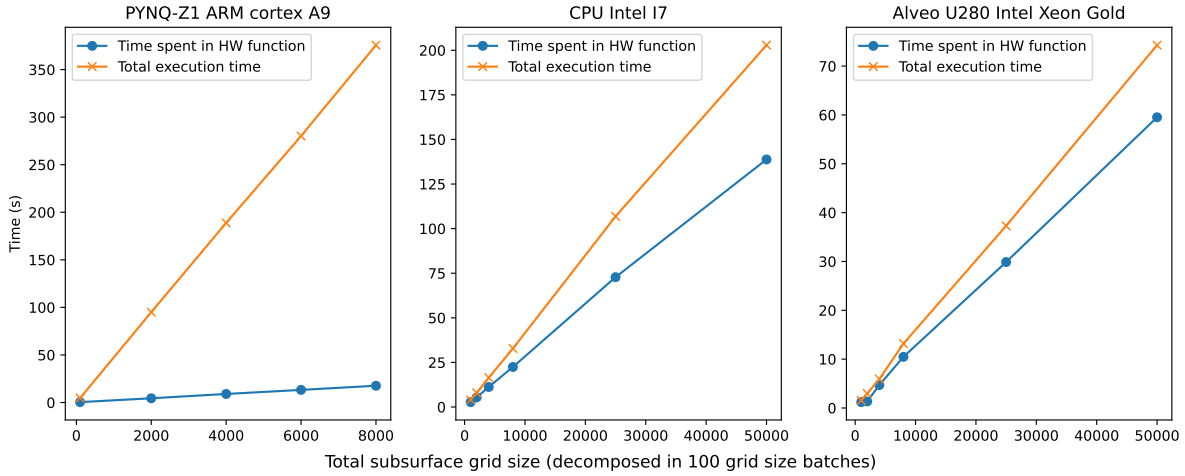

Figure 6.7: An overview of the execution time spent in the hardware-accelerated functions for the PYNQ-Z1 and Alveo U280 FPGAs and CPU python implementations. The high utilization PYNQ-Z1 hardware design from Table 6.1 with resolution 294 was used. The python FWI implementation was used for both CPU and PYNQ-Z1. These results do not take initialization and downloading the bitstreams to the FPGAs into account.

shows that the throughput established in the previous section for this FWI configuration on PYNQ-Z1 scales linearly with the number of nodes in the cluster. Because it scales linearly it is possible to extrapolate the throughput to discover how many PYNQ-Z1s are required to match the other hardware platforms in throughput. In the right plot, this exercise is visualized for the fastest hardware platform, the Alveo U280. The results show that 32 PYNQ-Z1 nodes are required to achieve the same throughput as a high-end Alveo U280 FPGA. To match the C++ and python CPU implementations respectively 22 and 12 PYNQ-Z1s are necessary.

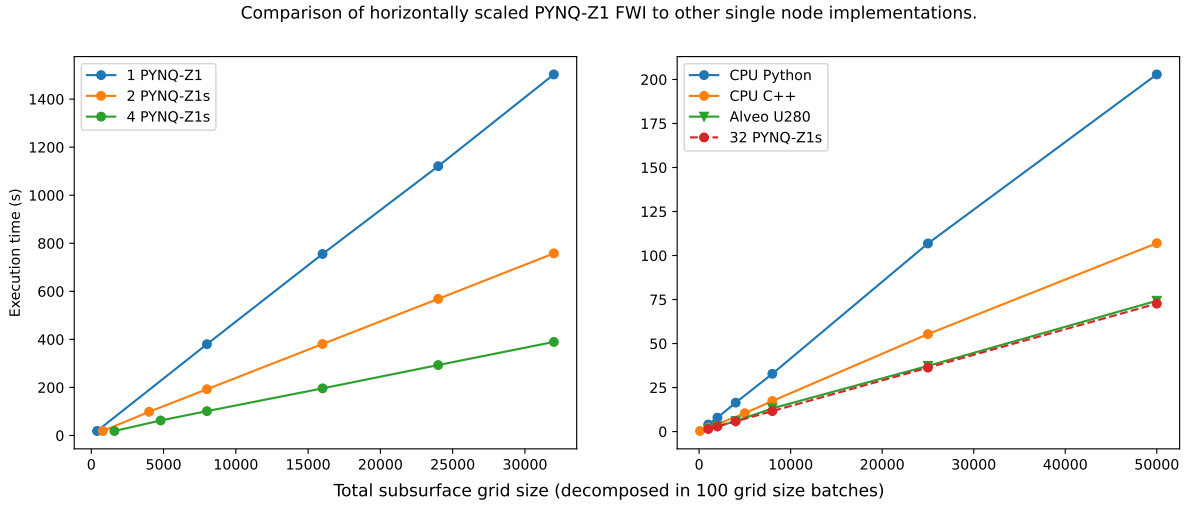

Figure 6.8: The left plot shows the performance of FWI with a resolution of 300 and grid size 100 when scaled on 1, 2, and 4 PYNQ-Z1s. The right plot shows the performance of the same FWI configuration for the python and C++ CPU implementation, the Alveo U280, and 32 PYNQ-Z1s.

The previous exercise was done to compare the total throughput of FWI on different hardware platforms. FWI is an application that is not fully accelerated in the PL. For each iteration in FWI, data is transferred back and forth twice to the forward function kernel and once to the cost function kernel. In general, transferring data between host memory and PL is detrimental to performance but necessary for this FWI hardware design. To compare the throughput of the hardware-accelerated functions on PYNQ-Z1 and Alveo U280 the throughput is computed under the assumption that the entire algorithm

is performed by the hardware functions. In other words, the time spent in hardware functions is taken as the total execution time. The same exercise is performed for the software part and the results are presented in Table 6.4. The table shows that the hardware throughput of the PYNQ-Z1, CPU baseline, and Alveo U280 are respectively 408, 359, and 934 grid cells per second. These numbers are displayed to show that for an application fully implemented in hardware the PYNQ-Z1 performs a factor of 1.36 better than the CPU baseline and only a factor of 0.44 worse than the Alveo U280. This means that by extrapolating, 3 PYNQ-Z1s are able to achieve the same performance as an Alveo U280. It should be noted that the comparison made is not entirely fair for several reasons discussed in Section 6.3.4, where other metrics of comparison between low and high-end FPGAs are presented.

Throughput (grid cells / second)	PYNQ-Z1	CPU	Alveo U280
Total	21	245	658
Software part	23	772	2861
Hardware part	408	359	934

Table 6.4: The throughput for different hardware platforms for an FWI configuration with a resolution of 300 and grid size of 100. The software and hardware throughput are computed under the assumption that their respective execution time was the total execution time.

Vertical scaling

The hardware design that was used in the previous section was bounded in resolution and grid size by the PYNQ-Z1 resources. The Alveo U280 FPGA has orders of magnitude more resources available which makes it possible to explore the impact of vertical scaling by using copied compute units. Vertical scaling is applying domain decomposition within one node, orthogonal to horizontal scaling where the domain is decomposed over multiple nodes, as was done for the PYNQ-Z1s in the previous section. In Figure 6.9 two plots are displayed to compare different combinations of vertical and horizontal scaling. In the left plot (A) from left to right, the first configurations consist of a single compute unit design that can process 100 grid cells on one node. The second configuration also processes 100 grid cells per node but divides the batches over two nodes. The third configuration processes 100 grid cells per compute unit and consists of two compute units in one hardware design on one node. The right plot (B), consists of the same configurations for resolution 1000.

From plots A and B a number of observations can be made. First, in both plots, the configurations applying vertical scaling have similar performance to the configurations with horizontal scaling. In plot A, the configuration with two compute units performs slightly worse than the configuration with one compute unit but two nodes. In plot B this is the other way around. In both hardware designs, the kernels are connected to memory banks DDR0 and DDR1. This means that the FPGA internally needs to schedule the execution of the compute units such that their data transfers do not interfere with each other. In plot A the resolution is much higher and consequently, the data transfers between the FPGA and host memory are larger than for the 1000 resolution design. A possible explanation that the higher resolution design with multiple compute units slightly under-performs compared to the lower resolution design is because the scheduling for data transfer between the two compute units and host memory is more complicated. Despite these small discrepancies, the configurations that apply vertical or horizontal scaling perform much better than the configurations that do not. An important conclusion is that the choice of FWI configuration relates to a trade-off between the quality of the results and the performance of the algorithm. The resources can be utilized to achieve high-quality results by using a high resolution or they can be used to achieve higher throughput by using multiple compute units with a lower resolution.

In summary the following conclusions were derived from the experiments in this section:

- Horizontal scaling of FWI results in a linear performance increase.
- Because FWI is only partially implemented in hardware, the type of host processor influences the total performance strongly.
- 32 PYNQ-Z1s are required to match the throughput of 1 Alveo U280 for the full algorithm. Considering only the hardware accelerated part of the algorithm, 3 PYNQ-Z1s outperform the Alveo

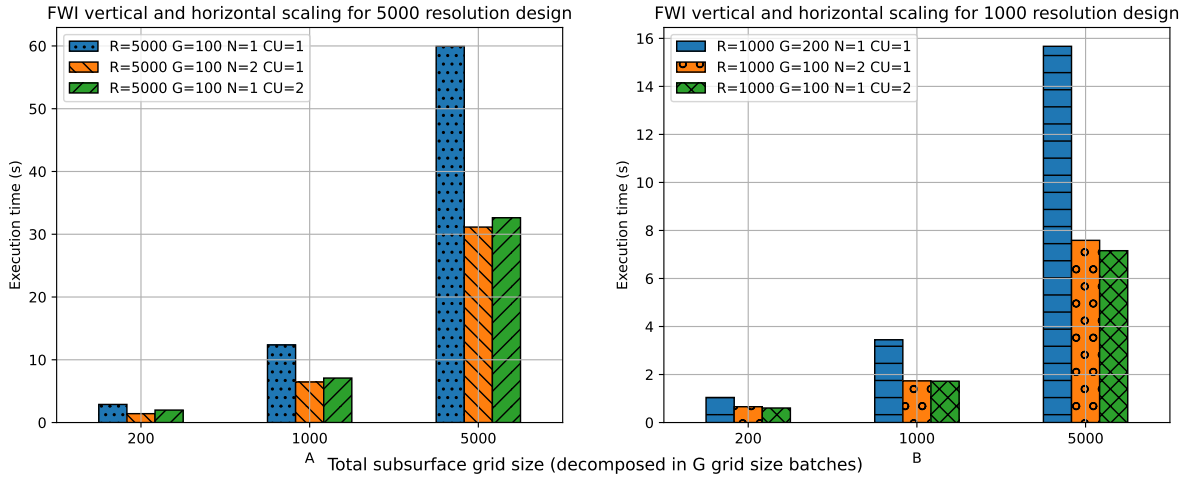

Figure 6.9: A comparison of vertical, horizontal, and no scaling for 1000 and 5000 resolution FWI configurations on Alveo U280. R denotes the resolution, G the batch of grid cells per compute unit or per node, and CU the number of compute units.

U280.

- The performance increases linearly when scaling vertically and horizontally. When scaling vertically, a trade-off is made between performance and accuracy, defined by the resources invested in resolution or decomposed local grids.

6.3.3. Performance

In this section, an FWI hardware design utilizing a high percentage of Alveo U280 resources is compared in terms of performance to the CPU baselines. In Figure 6.10 an overview is presented of the execution time of the python and C++ implementation on CPU and in combination with respectively OctoRay and pynq and OpenCL on Alveo U280. The FWI configuration processes 100 grid size batches with a resolution of 8000. The utilization of the hardware design is shown in the top row of Table 6.2.

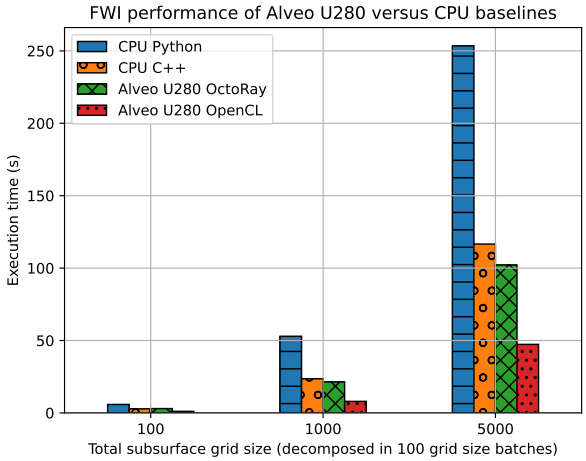

Figure 6.10: The performance of an 8000 resolution and 100 grid size FWI configuration. The comparison is made for the C++ and python implementation on CPU, the former in combination with OpenCL and the latter in combination with OctoRay and subsequently pynq, on an Alveo U280 FPGA.

The first observation is that the CPU baseline C++ implementation is approximately twice as fast as the python implementation. This effect is propagated in the Alveo U280 results as the OctoRay and therefore python implementation is approximately two times slower than the C++ OpenCL implementation. Additionally, the OctoRay implementation suffers from the overhead of transferring data back

and forth between the client and worker. The OpenCL Alveo U280 implementation attains the highest performance, being more than twice as fast as the CPU C++ implementation. It should be noted that in general high-end FPGA acceleration can result in much higher speed-ups. These higher speed-ups were not attained with the FWI hardware implementation due to several reasons. First and foremost, the hardware design was made by a software engineer with no prior experience in hardware design or FPGA development. HLS tools were used to achieve a working, but far from optimal hardware design. Secondly, FWI is not fully implemented in hardware and requires a significant part of the algorithm to be processed on the host processor. This also causes multiple large data transfers from the host processor's main memory to PL during each iteration of FWI. Data transfer is a common bottleneck in performance and should be avoided as much as possible. Consequently, there are many possibilities to improve the created hardware design. It should be noted that the focus of this work was not to implement the most optimal hardware design for FWI.

In Figure 6.11, the overhead induced by OctoRay and consequently Dask is displayed. Dask states that the overhead of scheduling a single task is approximately 1 millisecond [56]. The rest of the overhead is due to the data transfers between the client and workers. The total overhead consists of respectively 0.45%, 0.081% and 0.037% of total execution time for the 100, 1000 and 5000 input sizes. The overhead of scheduling is minimized by only scheduling a single task for the sum of batches and decomposing them locally.

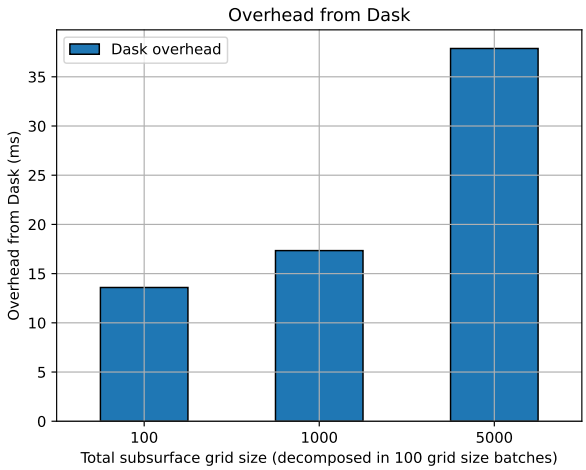

Figure 6.11: The overhead in milliseconds induced by Dask for 100, 1000 and 5000 total subsurface grid sizes executed by an 8000 resolution 100 grid size FWI configuration.

6.3.4. Cost and utilization

In the previous section, the performance of several FWI configurations scaled with different techniques was presented. To provide a better comparison of the different hardware platforms not only the performance should be taken into account, but also the cost of the platform and whether the full potential of the platform is utilized. In this section, the focus lies on comparing the low-end and high-end FPGAs in terms of cost and utilization.

In Table 6.5 an overview is given of the prices, at the time of writing, for the different hardware platforms targeted in this work. Additionally, the extrapolated prices for the number of PYNQ-Z1s to match the C++ CPU implementation and the Alveo U280 implementations when considering the total throughput and only the hardware throughput are presented. It should be noted that to utilize the PYNQ-Z1s in parallel with OctoRay, a local network consisting of high throughput switches and ethernet cables is also required that were described in Section 6.1.2. A standalone CPU also requires RAM and other components to process the algorithm, which is why the price of the laptop containing the CPU has been used in this comparison.

In the throughput comparison of PYNQ-Z1s, the baseline CPU, and Alveo U280s, the hardware throughput was introduced for a theoretical application of FWI where the entire algorithm is executed in the PL. This theoretical scenario is introduced to be able to isolate and compare the hardware acceleration of the FPGAs without interference from the host processor's performance. The presented prices in Table 6.5 show that a PYNQ-Z1 setup consisting of 32 nodes costs about 2000 dollars less than an Alveo U280 in combination with a server-grade processor. This number suggests that the PYNQ-Z1 setup could be a viable alternative for this FWI configuration. It is important to understand that this comparison is made for a hardware design that utilizes nearly the full resources of PYNQ-Z1, while only a fraction of the Alveo U280. Additionally, two laptops at a price of 2200 dollars would outperform both the 32 PYNQ-Z1s and the Alveo U280, but also while fully utilizing its resources.

	Alveo U280	**HP Zbook studio G5 (Intel I7 8750H CPU)**	**PYNQ-Z1**	**3 PYNQ-Z1s**	**22 PYNQ-Z1s**	**32 PYNQ-Z1s**
FPGA ($)	8.145,40	-	252,04	3 x 252,04	22 x 252,04	32 x 252,04
Processor ($)	2133,79	1099				
Total ($)	10.279,19	1099	252,04	756,12	5544,88	8065,28

Table 6.5: Cost overview in dollars of Alveo U280 and Intel Xeon Gold host processor, PYNQ-Z1 including dual-core cortex A9 host processor and an Intel I7 8750H processor. The prices displayed were the cheapest available in the Netherlands at the time of writing.

Table 6.2 presents an overview of the percentage of the total utilization used for different FWI configurations for Alveo U280. The bottom row of the table shows the percentage of resources used for the design allowing for a maximum resolution of 300 while processing 100 grid-size batches. This was the design used in the throughput compared with the other platforms. Clearly, this design utilizes only a fraction of the total resources available in the Alveo U280.

To provide an understanding of the resources utilized by the different hardware setups, Table 6.6 shows the total available resources and the factor of difference for PYNQ-Z1 and Alveo U280 in the first four rows. The bottom four rows show the utilized resources for the FWI design used in the previous comparisons on different hardware setups. A complete overview of the utilization of all FWI hardware designs on Alveo U280 was shown in Table 6.2. The utilized resources numbers show that even though using 32 PYNQ-Z1 boards can match the total throughput of an Alveo board for an FWI design with relatively low resolution, the total amount of resources is much higher than that of the Alveo. This makes sense as the Alveo applies sequential domain decomposition processing batches of 100 grid cells one after the other utilizing the same resources for each batch. On the contrary, the PYNQ-Z1s process the grid batches in parallel using a set of full resources for each grid batch. The bandwidth from the PL to the main memory on an Alveo U280 card is orders of magnitude higher than than the PYNQ-Z1 bandwidth shown in Table 2.1. Additionally, the multiple PYNQ-Z1 setups suffer from overhead when transferring data over the ethernet cables and switches to disperse and merge the decomposed grids. When considering the theoretical scenario of FWI fully accelerated in PL, only 3 PYNQ-Z1 nodes are required to match the Alveo U280 in hardware throughput, using a similar amount of memory resources, half the LUTs and a fifth of the FFs.

Available resources	**LUT (x1000)**	**FF (x1000)**	**DSP**	**BRAM**	**URAM**
PYNQ-Z1	53.2	106.4	220	280 (18Kb)	**-**
Alveo U280	1304	2607	9024	2016 (36Kb)	960 (288Kb)
Factor	24.5	24.5	41	14.4	-
Utilized resources for FWI R=300 G=100					
1 PYNQ-Z1	26.6	17.03	33	4.7 (Mb)	0
3 PYNQ-Z1s	79.8	51.01	99	14.1 (Mb)	0
32 PYNQ-Z1s	851.2	545	1056	150.4 (Mb)	0
Alveo U280	183	287	90	12.3 (Mb)	11.1 (Mb)

Table 6.6: An overview of the total available resources and the utilized resources for 1, 3, and 32 PYNQ-Z1s as well as 1 Alveo U280 FPGA for an FWI hardware design with resolution 300 processing 100 grid size.

6.3.5. Design-build duration

In the previous section low and high-end FPGAs were compared in terms of cost, utilization, and throughput. Another metric of comparison is the design-build duration. As explained in Chapter 2, the design flow for FPGAs is a complex, time, memory, and computationally demanding process. Because each step of the design flow is prone to errors, verification and simulation are performed after each phase. Although these methods help trace initial errors, some errors are only discovered when testing the final bitstream. Therefore, the design-build duration forms an important metric for the development cycle and consequently the total time to market an application. To provide a fair comparison, Figure 6.12 shows the design-build duration of FWI designs that utilize a high percentage of their respectively targeted platforms' resources. The figure shows the time spent to synthesize the kernels and the time spent in the rest of the design-build. This differentiation was made because during HLS hardware design the conversion from software written kernels to synthesized kernels is an important target where a developer can validate the correctness of the kernel code. In general, this phase is iterated over more often than the entire design-build duration during the application development cycle. The figure shows that an 8000 resolution hardware design for Alveo U280 takes around 4 and a half hours and a 300 resolution design takes 3 hours and 15 minutes. These designs require a total build duration of respectively 9 and 6 times more than the PYNQ-Z1 total build duration of 35 minutes. It should be noted that the PYNQ-Z1 bitstream was built on an HP Studio Zbook G5 personal laptop while the Alveo U280 bitstream was built on the qce-alveo01 server from TU Delft containing 40 server-grade logical cores and 96 GB of RAM.

These numbers show that the development cycles of applications for a low-end FPGA can be performed quicker than for high-end FPGAs. It should be noted that a generated bitstream for a certain type of FPGA can be copied to any number of FPGAs of the same type. Therefore the deployment of an application on a multiple PYNQ-Z1 setup only requires the extra effort of setting up the distributed execution with OctoRay. The improvements to OctoRay were important in order to reduce the effort of setting up distributed execution to a minimum and make it reusable for different configurations and applications.

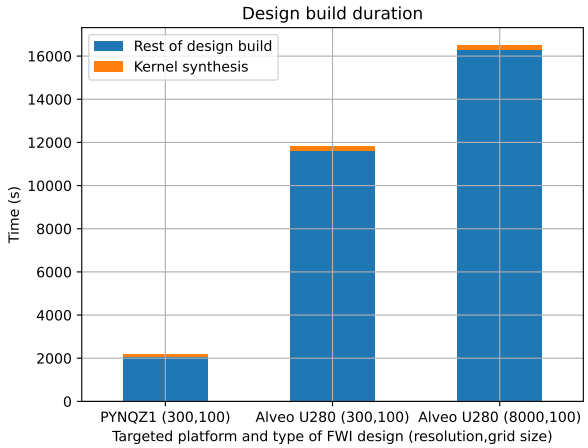

Figure 6.12: The total design-build duration is divided into kernel synthesis and the remaining build duration for three hardware designs of FWI.

6.3.6. Power

Figure 6.13 displays the global trends in internet traffic, data center workload, and energy consumption from the International Energy Agency (IEA). The trends are relative to 2010 but clearly show that although internet traffic and data center workload has increased considerably in the last decade, the energy consumption has only slightly increased. It should be noted that these are global trends. A European Commission study shows that in Europe data centers accounted for 2.7% of the electricity demand in 2018, with energy consumption of 76.8 TWh. The absolute increase in data center energy consumption is expected to grow by 28% to 98.52 TWh while the relative increase of electricity demand will reach 3.21% by 2030 [57].

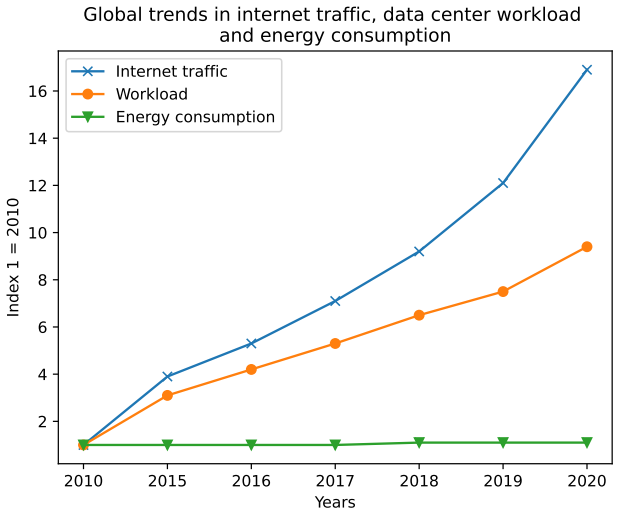

Figure 6.13: The global trends in user internet traffic, data center workload, and energy consumption. The numbers in this graph are relative to 2010. Data was taken from IEA [58]

The reason energy consumption is not increasing linearly with workload and internet traffic is due to the developments in energy-efficient hardware and software as well as physical improvements in the cooling systems of data centers [59]. From the European expected trends, it can be concluded that the growth in workload is outpacing the developments in energy efficiency. These trends emphasize the importance for hardware and software developers to be conscious of their application's and implementation's energy consumption.

In Figure 6.14 the power and energy consumption for different hardware platforms and FWI configurations are displayed. Power was measured with the setup from University of Kassel for the PYNQ-Z1 designs with resolution of 300 and grid sizes of 100 and 20. The setup only measured the power of the PL and not of the ARM core. The Alveo U280 power was measured with the internal sensor and the xbutil tool that comes with XRT, which also does not measure the host processor power consumption. Because of this, the PYNQ-Z1 extrapolated numbers for 3 and 32 devices were based on the measured power of the design with resolution 300 and grid size 100. The energy consumption is calculated in Joules per 100 processed grid cells. Instead of using the total execution time to calculate the energy, the time spent in the hardware functions was used because the power measurements are based on the power consumed in the hardware functions.

It can be observed that an increase in total grid size does not affect the power consumption for the FPGAs. This makes sense because in these measurements the total grid size was increased but the algorithm processes the total grid in batches of the same size. Therefore the computational complexity and memory demand for each batch are approximately the same resulting in constant power consumption. Because the power consumption remains constant, independent of the grid size, the extrapolation to 32 PYNQ-Z1s in terms of power is possible. In practice, the full throughput of the PYNQ-Z1s is only achieved when they process 100 grid cells per node, to a total of 3200. Additionally, dividing a total of 100 grid cells over 32 nodes by processing 3 grid cells per node does not make sense in practice because a minimum grid size is necessary to achieve reliable results. Nevertheless, with the observation of constant power regardless of total grid size, it is possible to compare the PYNQ-Z1s and the other hardware platforms. Another observation is that the impact of a lower or higher utilization is very small on the power consumption for PYNQ-Z1 and Alveo U280. The Alveo U280 power measurement accuracy is more coarse-grained than the PYNQ-Z1 setup which is why the difference, less than 1 Watt, could not be measured. It should be noted that the idle power consumption of the Alveo boards is only 2 Watts less than the measured power during execution, which means that the FWI hardware designs require little power. One of the explanations for this is that a significant part of the algorithm is executed on the host processor of which there are no power measurements.

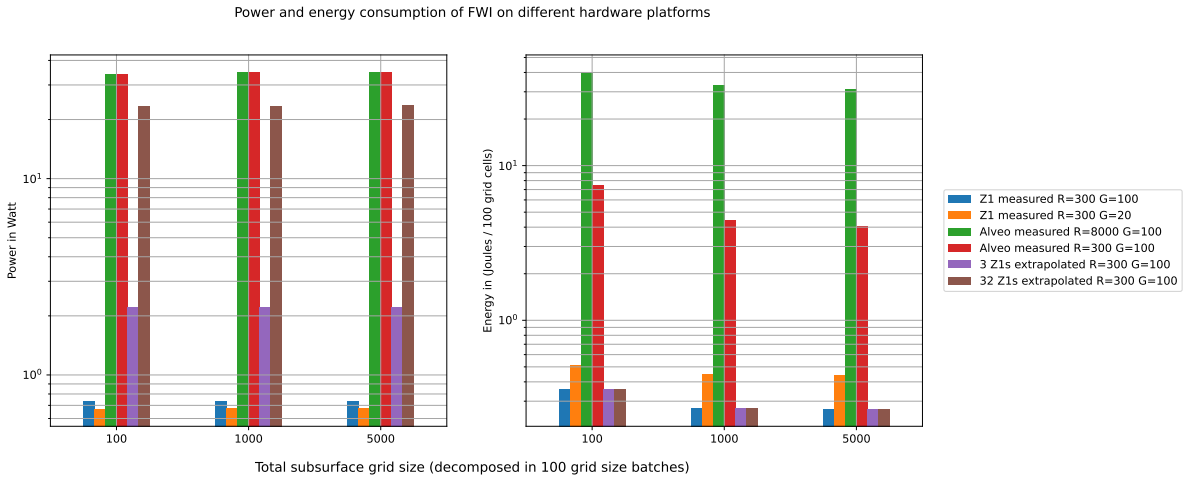

Figure 6.14: The power and energy consumption of the PYNQ-Z1, and Alveo U280 FPGAs. The power is measured for total grid sizes 100, 1000, and 5000, decomposed in batches of 100 grid cells. The energy is calculated by using the time spent in the accelerated part of the algorithm as the power is also only measured over that part.

Because the power remains approximately constant irrelevant of resolution and grid size, but the execution time decreases for lower utilization designs, the energy consumption per 100 grid cells also decreases for these designs. The higher resolution designs take longer to process batches of 100 grid cells consequently resulting in higher energy consumption. This can be seen in the figure when comparing the 8000 and 300 resolution configurations for FWI on Alveo U280 and the 100 and 20 grid size designs for PYNQ-Z1. The energy consumption for a total grid size of 100 is slightly higher than for the 1000 and 5000 grid sizes. This is because when processing one batch, the overhead of initialization is relatively higher than when processing multiple batches. Because the power is measured for the hardware functions, conclusions about power consumption can only be made for the theoretical scenario that FWI is entirely accelerated in hardware and does not require a host processor for computations. In this scenario, the Alveo U280 consumes 4.1 Joules per 100 grid cells which is approximately 15 times more than the PYNQ-Z1s at 0.27 Joules per 100 grid cells. The 8000 resolution design consumes 31 Joules per 100 grid cells, which is 115 times more than the PYNQ-Z1.

In summary, the following conclusions were derived from the experiments in this section:

- The power was only measured for the PL.
- Power consumption is only minimally affected by the resolution and grid size.
- Higher resolutions lead to higher energy consumption as the execution time increases and the power remains constant.

6.4. CNN results

This section discusses the results from the CNN experiments with OctoRay on the HACC cluster's Alveo U280 FPGAs. The application was included in this work to compare a multi-input dependent application to a single-input dependent application in terms of scalability. There are two important differences between CNN and FWI. First, unlike FWI, CNN is completely implemented in hardware and only requires the host processor as an interface to provide input data and retrieve output data. Next, the size of the hardware design grows based on the input for FWI inversion, while it does not for CNN. In other words, to implement an FWI design for a higher resolution or grid size, more resources of the FPGA are necessary.

The CNN is different in this regard, as the input is always in the same format the used resources are independent from the number of inputs. Because a single instance of the CNN does not utilize all the resources of the Alveo U280, multiple instances of the CNN can fit into a single hardware design, which is referred to as vertical scaling.

6.4.1. Scalability

In this section, the effect of vertical and horizontal scalability for the CNN application is displayed and discussed. Vertical scaling is applied by introducing multiple instances of the CNN in one hardware design, where horizontal scaling is performed with the help of the OctoRay framework by scaling on multiple Alveo U280 nodes. To ensure the most reliable results of scalability, an analysis is performed to find the input size for which the maximum throughput of a single instance hardware design can be reached. In Figure 6.15, the throughput in images per second is displayed for different batch sizes in the OctoRay setup with a single instance hardware design and 1 dask worker. The worker is located on a different machine than where the client is run, as specified in the caption.

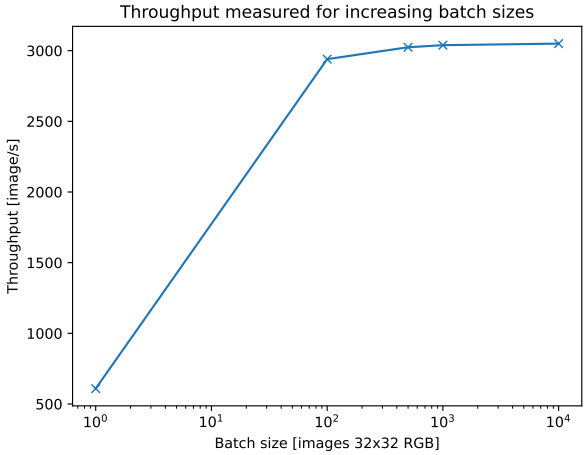

Figure 6.15: The throughput in images per second of 1 dask worker executing the CNN application with a 1 instance hardware design for batch sizes 1, 100, 500, 1000, and 10000 of 32x32 RGB images. The dask scheduler and client were on the HACC alveo3b with IP 10.1.212.126 and the dask worker on alveo3c with IP 10.1.212.127.

From Figure 6.15 it can be seen that the throughput of a single instance hardware design on one node reaches its limit at approximately 3050 images per second. For a batch size of one thousand images, the reached throughput is only 12 images per second slower than the limit of 3050 images per second. In order to correctly analyze the benefits of vertical and horizontal scaling a batch size larger than 1000 per compute unit is used such that the maximum throughput is reached.

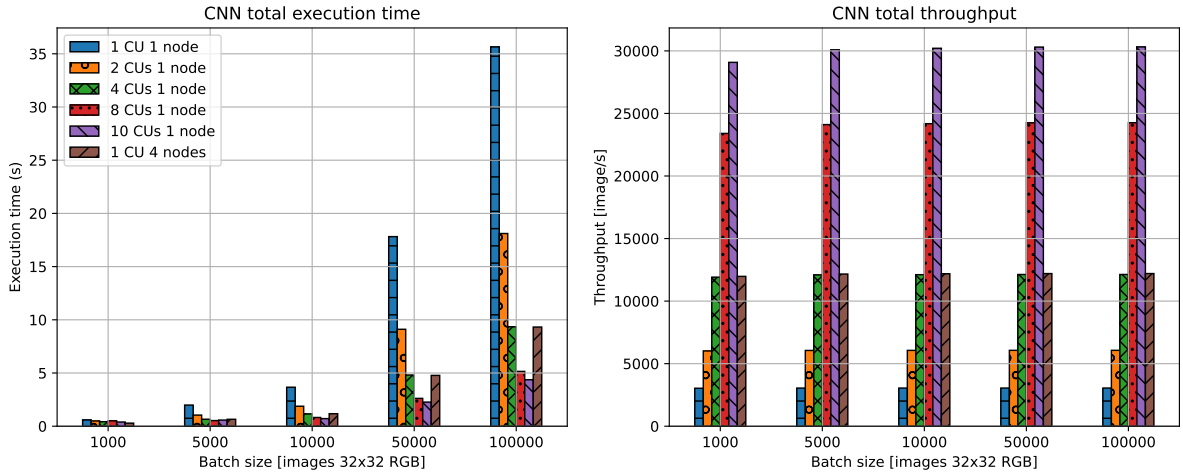

Figure 6.16: The total execution time and throughput for CNN designs with 1,2,4,8 and 10 CUs on 1 node with 1 dask worker and a CNN design with 1 CU on 4 nodes and dask workers. The numbers represent the average of 5 runs for batch sizes 1000, 5000, 10000, 50000, and 100000 of 32x32 RGB images from the cifar-10 dataset.

In Figure 6.16 the left plot displays the total execution time and the right plot the total throughput for 5 different OctoRay configurations on one node, and one configuration on 4 nodes. The horizontal axis displays the batch sizes of the experiments. It should be noted that the total batch size is displayed, which means that a design with 4 compute units processes 250 images per compute unit for a total batch size of 1000 images. Consequently, the compute units do not reach the maximum throughput until the total batch size is 1000 fold of the amount of compute units. From the right plot in Figure 6.16, it becomes clear that the total throughput scales linearly with the amount of compute units present in a design. The total throughput achieved from horizontal scaling of a single instance design on four nodes is equal to that on a vertically scaled node containing four CNN instances.

6.4.2. Build design time

In this section, the development cycle to produce CNN designs with different numbers of instances is discussed. The CNN was built with the FINN compiler, unlike FWI that was manually implemented. The CNN was implemented as a BNN, which brings several advantages. First, FPGAs are known to perform orders of magnitude better on binary operations compared to floating point operations. Therefore, quantizing the weights and activations to 1 or 2 bits instead of using floating points can benefit the performance greatly. As explained in Section 3.2.1, high accuracy can still be attained with 1 or 2 bit quantization. In addition to the performance benefits, using 1 or 2 bits instead of floating points also requires much less memory resources. In Figure 6.17, the design build duration of hardware designs containing different numbers of CNN instances is shown. The duration of the different stages of the build are specified in the bars. From the figure it can be read that a single instance CNN takes approximately 2 hours and 5 minutes, the other extremity of 10 instances takes 5 hours and 23 minutes. Clearly, the design build duration does not increase linearly.

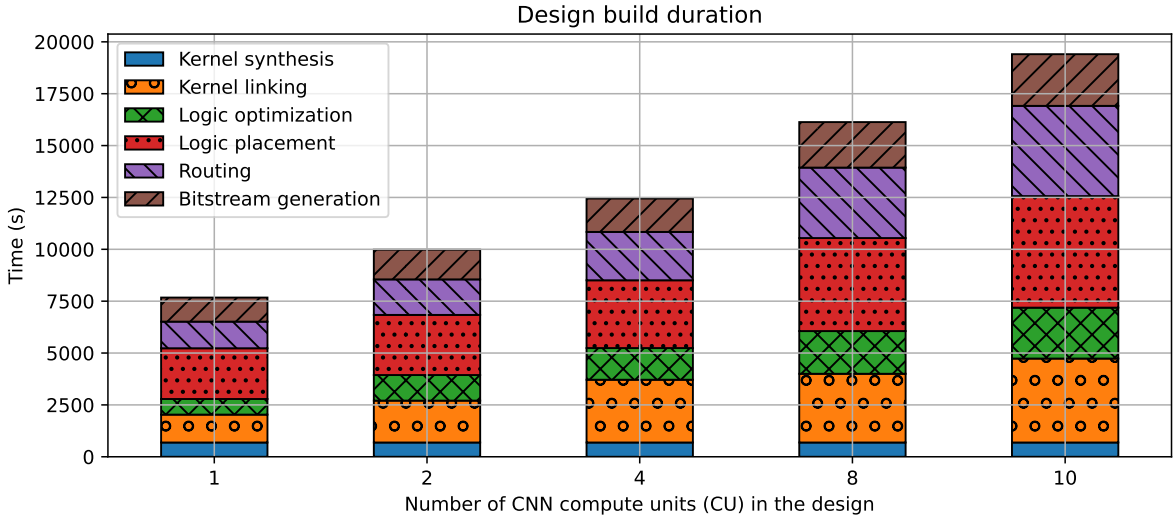

Figure 6.17: The design-build duration for 1, 2, 4, 8, and 10 CU CNN designs decomposed into the different components of the total design-build.

When creating designs with multiple instances, the kernel synthesis only needs to take place once which is why it stays constant. In Figure 6.18 the percentage of the total build duration is specified for the different phases of the build. Because the kernel synthesis duration stays the same but the total build duration increases, the relative time spent in this phase decreases for designs with multiple instances. Additionally, it can be observed that especially the kernel linking and routing phases are relatively more time consuming for designs with multiple compute units. This is an expected result as linking more kernels is complexer than fewer kernels. More kernels also means more resources of the FPGA are used and further spread over the board. Amongst other things, this makes the routing of interconnections a more challenging and time consuming task. It should be noted that the design build duration was measured on the qce-alveo01 server from TU Delft. As this is a shared server, the load may have varied depending on other users' activity and affected the results.

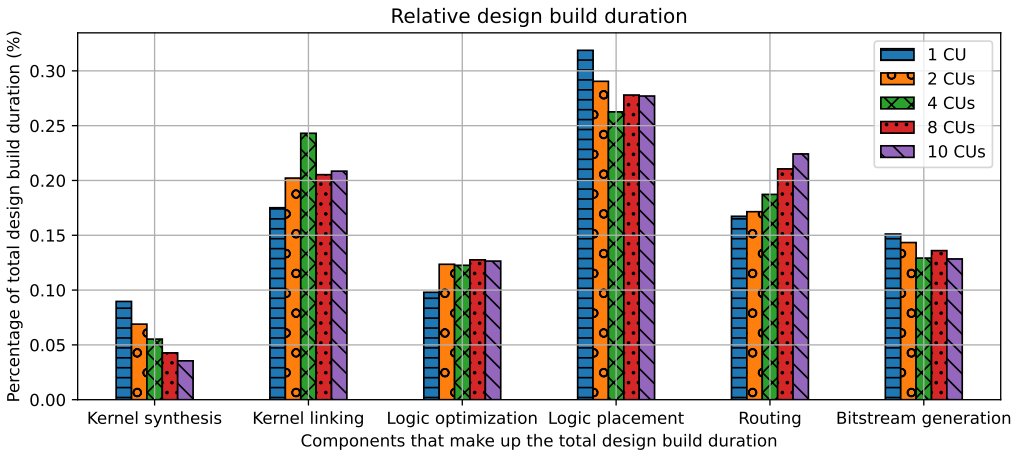

Figure 6.18: The throughput in images per second of 1 dask worker executing the CNN application with a 1 instance hardware design for batch sizes 1, 100, 500, 1000 and 10000 of 32x32 RGB images.

7

Conclusions and future work

7.1. Conclusions

In this work a twofold investigation was performed. To improve the accessibility and aid the adoption of FPGAs the OctoRay framework was developed that enables users to transparently scale applications horizontally and vertically on FPGAs. Additionally, an FPGA implementation for FWI was developed and used to investigate the viability of a distributed low-end FPGA composition alternative to a single high-end FPGA. In this chapter the conclusions from these investigations are used to answer the defined research questions.

1. How can FPGAs be used to scale data analytics applications in a transparent and accessible way?
2. Can FWI be accelerated and scaled on FPGAs?
3. Is a composition of low-end FPGAs a viable alternative to a data center-class FPGA in terms of performance, power efficiency, and cost when accelerating FWI?
4. How does vertical and horizontal scaling affect the performance of FPGA accelerated applications?

7.1.1. Accessibility

The scalability of data analytics applications on FPGAs was addressed by developing the OctoRay framework that enables users to transparently scale applications. The framework was extended from an initial collection of Jupyter notebooks that demonstrated how the parallelization framework Dask could be used to scale applications on multiple FPGAs. The new architecture implemented into a library in this work led to following improvements.

1. **Automatic deployment:** To improve the usability and large scale deployment of OctoRay the framework now implements automatic SSH deployment of a Dask cluster over any number and type of nodes.
2. **OctoRay kernels:** The initially rigid architecture was directed at scaling a single application on same type FPGAs. OctoRay kernels were implemented to incorporate the specifications and configurations of an application in a single entity. This allows users to create and execute endless application types and configurations with minimal effort. Internal helper functions were developed to handle the deployment of OctoRay kernels on the cluster. Most importantly, OctoRay kernels form a layer of abstraction that automatically handles vertical and horizontal scaling without extra user effort. The cluster and worker deployment for an OctoRay kernel that scales a single instance hardware design over 10 nodes or a ten instance design on one node is completely handled internally.
3. **Library:** To improve the accessibility and ease of use, the OctoRay framework was packaged into a library and made available on the PyPI index making it pip installable.

7.1.2. FWI acceleration

This work was performed in a collaboration with ALTEN Netherlands and the ABS research group from TU Delft. ALTEN provided a C++ implementation of the FWI application with the task to accelerate it on an FPGA. Therefore, the FPGA design flow was performed to develop hardware designs for the PYNQ-Z1 and Alveo U280 FPGAs. Different optimization techniques were applied to utilize the FPGA resources efficiently. These efforts led to a FWI hardware design with resolution 300 and grid size 100 for a low-end PYNQ-Z1 FPGA and an 8000 resolution and grid size 100 design for the high-end Alveo U280 FPGA. Different solution architectures were developed that consisted of python and C++ host code and PYNQ and OpenCL hardware communication layers. It was concluded that the baseline python implementation performs 2 times slower than the C++ implementation. This effect propagated in the FPGA implementations, where the OpenCL and C++ implementations performance was measured at twice the PYNQ and python implementation. The latter performed similarly to the C++ CPU baseline. In general, accelerating an application on a high-end FPGA should result is much higher speedups compared to a regular personal laptop. For ease of use and reproducability, HLS code was used to implement the hardware designs. This has its benefits but does not properly harness the capabilities of the FPGA. In addition, FWI was only partially implemented in hardware which led to two consequences. First, the performance of the host processor had a significant impact on the total performance. Secondly, each iteration of FWI required multiple data transfers between the host processor main memory and the PL of the FPGA, which is detrimental to performance. Conclusively, a higher performance was achieved by accelerating FWI on a high-end FPGA but with the current level of optimization a CPU implementation is a more cost-efficient alternative.

7.1.3. Cost-efficient compositions

Data center class FPGAs such as the Alveo U280 are powerful machines that can attain enormous performance, throughput and latency boosts compared to traditional CPUs. Alternatively, under the condition that data parallelism can be applied, a cluster of low-end FPGAs could be used as an alternative composition. After having established that domain decomposition could be used to divide a total subsurface grid into independent local domains it became possible to apply data parallelism to invert the local domains in parallel on PYNQ-Z1 low-end FPGAs.

It was shown that 32 PYNQ-Z1 devices, executing in parallel, are required to achieve the throughput of a single Alveo U280 FPGA for a FWI configuration with a resolution of 300 and grid size 100. The multi-node PYNQ-Z1 setup is approximately 20% cheaper than the data center class alternative. As only a fraction of the Alveo U280s resources are required for this FWI implementation, the full potential of the accelerator is not harnessed. Moreover, it appeared that two regular laptops outperform the Alveo and multi-node setup at a quarter of the Alveo's price.

To extend the analysis, a theoretical scenario was used that compared the hardware platforms solely based on the hardware accelerated part of FWI. In such a scenario 3 PYNQ-Z1s outperformed the laptop and achieved 44% of the Alveo U280s throughput in grid cells processed per second. From the power measurements it was shown that PYNQ-Z1 boards consume 15 times less energy than an Alveo U280 when processing a 300 resolution configuration.

From these results it is possible to conclude that for applications that do require only a portion of the full potential of high-end FPGAs, rapid development cycles and low energy consumption, a device composition of multiple low-end FPGAs can be a viable alternative. Full Waveform Inversion was characterized as a multi-input dependent algorithm of which the quality of the results is mostly dictated by the resolution. In this work, experiments were performed on synthesized subsurface grid data sets with only homogeneous velocities. The non-linear properties over the entire subsurface grid were not taken into account after applying domain decomposition. Although the implementations in this work can not directly be applied in practice, they do show the potential of FPGA acceleration and scaling for FWI. Additionally, this allowed to investigate the effects of deploying a multi-input dependent application on different FPGA compositions.

7.1.4. Scalability

The CNN application was introduced to this work to compare the potential of horizontal and vertical scaling applied to the single-input dependent CNN and the multi-input dependent FWI.

The scalability of multi-input dependent FWI was analyzed by comparing the performance of vertical scaling, horizontal scaling, and no scaling. To compare these scaling techniques, three different hardware designs were used to invert a total subsurface grid of 200 cells at a resolution of 1000. The vertical scaling setup consisted of a single Alveo U280 node implementing a hardware design with two compute units each processing 100 grid cells in parallel. The horizontal scaling setup consisted of two Alveo U280 nodes each implementing a hardware design with one compute unit processing 100 grid cells. The third setup without scaling consisted of a single Alveo U280 implementing a hardware design processing the entire subsurface grid of 200 cells. The domain decomposed vertical and horizontal scaling setups performed similarly and showed a speed up of 2 compared to the setup inverting the entire subsurface grid. From these results it can be concluded that domain decomposition allows for parallel execution of smaller grid sizes and the execution of the smaller grids is faster. As long as the resources allow it, vertical scaling is a more efficient way of scaling as it does not require additional nodes. In the case of FWI, a trade off has to be made for what purpose the resources are used, either to improve quality of the results by increasing the resolution or in the performance of the algorithm by investing in multiple compute units.

For the CNN application there is no such trade off between quality and performance. Therefore, the throughput of the algorithm is dictated by the amount of compute units fitting on a single hardware design and the number of the nodes scaled on. The results showed that the total throughput in images classified per second scaled linearly for either type of scaling, horizontally on nodes and vertically with multiple compute units.

7.2. Future work

In this work an FPGA implementation of FWI was developed. Additionally, OctoRay was expanded to a scalability framework for FPGAs. This project was carried out in a specific time frame, bounding the possible research directions that could be further explored. Therefore a number of recommendations and idea's for future work are provided below.

- FWI was only partially implemented for an FPGA. By fully implementing the algorithm in hardware, the performance loss caused by the large and often occurring data transfers between the host main memory and the PL could be mitigated by the enormous internal bandwidth of FPGAs. It should be noted that fully implementing the algorithm in hardware would require more resources and consequently decrease the possible resolution and grid size processed. An analysis could be performed to investigate the viability of fully implementing FWI in hardware. Additionally, an interesting point of comparison would be a GPU implementation of FWI.

- The kappa matrix can attain very large proportions for high resolutions and grid sizes. In the current implementation, the matrix was kept in main memory and transferred to the PL for each kernel computation. Because the kappa matrix does not change over the course of the iterations, this results in a highly redundant data transmission. The current hardware design could be improved by reducing these redundant transmission by storing the kappa matrix in PL after the first transmission.

- As mentioned before, FPGAs attain much higher peak performance when processing binary instead of floating point operations. An investigation of quantizing the pressure field data from floating point to fixed point was performed for the synthetic data sets. The investigation produced a distribution for the temple data set and concluded that a 4 bit integer with precision 27 and 1 sign bit could be used to achieve an acceptable accuracy. As the synthetic data set consisted of only two different homogeneous velocities, these results are unreliable and left out of this work. A future project could analyze the distribution of real data sets and potentially improve the FPGA acceleration by applying fixed point quantization.

- From the FWI performance results it was shown that the OpenCL and C++ implementation performed significantly better than the python and pynq implementation. OctoRay could be extended to support the execution of C++ and OpenCL binaries.
- The OctoRay framework currently provides several CNN and FWI bitfiles for out-of-the box usage. To further increase the range of usability, a collection of bitfiles for popular applications could be made available such that they can easily be scaled up with OctoRay. A concept similar to the Model Zoo available in the machine learning field.

A

Full Waveform Inversion

A.1. Cost Function C++

Listing A.1: Cost Function

```
1  void ConjugateGradientInversion::getUpdateDirectionInformation(
2          const std::vector<std::complex<double>> &residualVector,
3          core::dataGrid2D<std::complex<double>> &kappaTimesResidual)
4      {
5          int l_i, l_j;
6          kappaTimesResidual.zero();
7
8          core::dataGrid2D<std::complex<double>> kDummy(_grid);
9          auto kappa = _forwardModel->getKernel();
10         for(int i = 0; i < _frequencies.count; i++)
11         {
12             l_i = i * _receivers.count * _sources.count;
13             for(int j = 0; j < _receivers.count; j++)
14             {
15                 l_j = j * _receivers.count;
16                 for(int k = 0; k < _receivers.count; k++)
17                 {
18                     kDummy = kappa[l_i + l_j + k];
19                     kDummy.conjugate();
20
21                     kappaTimesResidual += kDummy * residualVector[l_i+l_j+
                           k];
22                 }
23             }
24         }
25     }
```

A.2. Forward Function C++

Listing A.2: Forward Function

```
1  void FiniteDifferenceForwardModel::applyKappa(const core::dataGrid2D<
       double>
2  &CurrentPressureFieldSerial, std::vector<std::complex<double>> &kOperator
       )
3      {
4          for(int i = 0; i < _freq.count * _source.count * _receiver.count;
               i++)
5          {
6              kOperator[i] = dotProduct(_vkappa[i],
                   CurrentPressureFieldSerial);
7          }
8      }
```

A.3. Implementation diagram

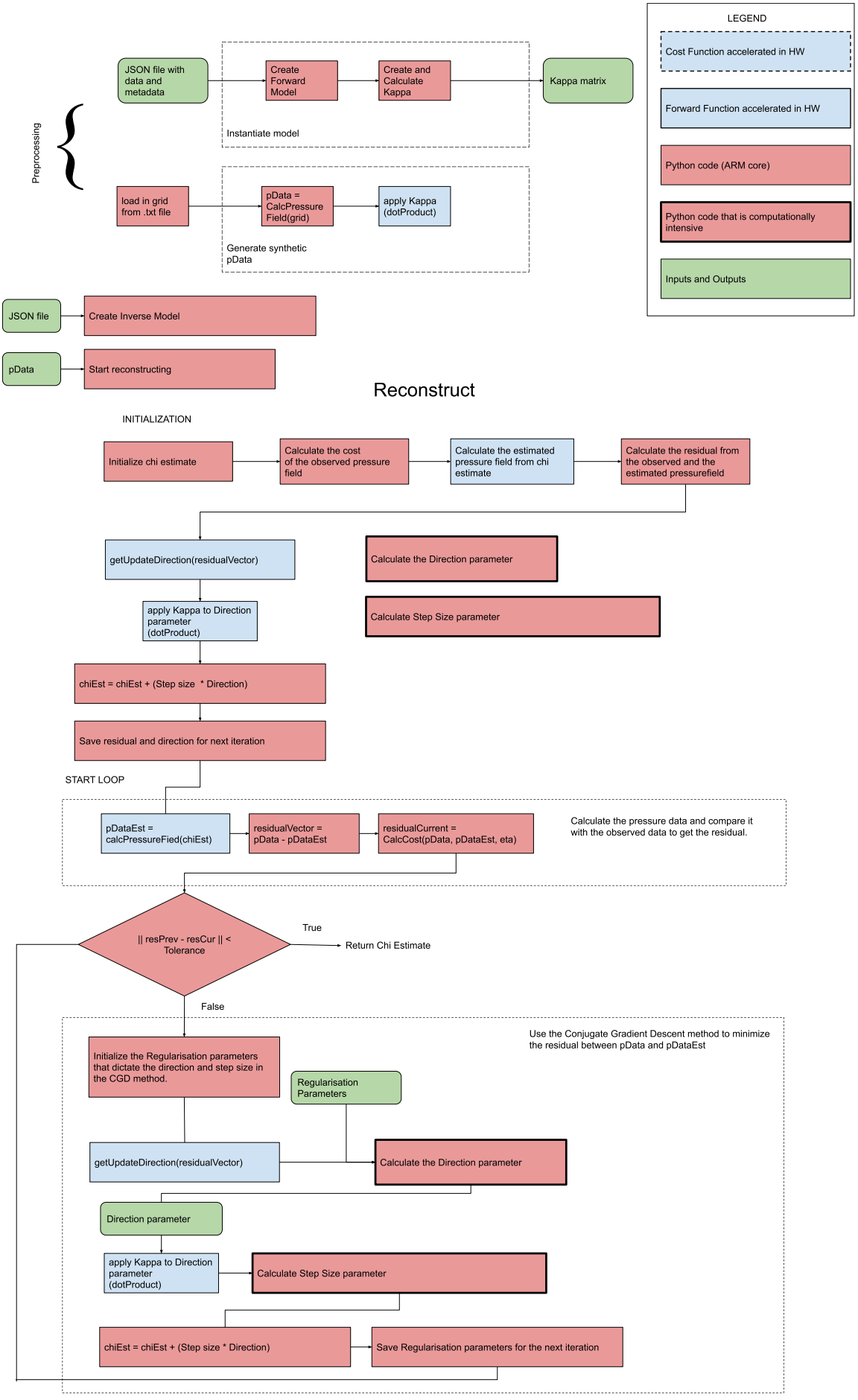

Figure A.1: A diagram of the ALTEN provided implementation of FWI using the finite difference forward model and a conjugate gradient descent inversion model.

Bibliography

[1] VITISLIB, *Vitis Libraries*, en. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html (visited on 06/02/2022).

[2] Blackmist, *Deploy ML models to FPGAs - Azure Machine Learning*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service (visited on 05/11/2022).

[3] E. J. Houtgast, V.-M. Sima, and Z. Al-Ars, "High Performance Streaming Smith-Waterman Implementation with Implicit Synchronization on Intel FPGA using OpenCL," en-US, Nov. 2017. [Online]. Available: https://hgpu.org/?p=17808 (visited on 06/26/2022).

[4] J. Fang, J. Chen, Z. Al-Ars, H. Hofstee, and J. Lee, "An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic," *Journal of Signal Processing Systems*, vol. 92, pp. 1–17, Sep. 2020. DOI: 10.1007/s11265-020-01547-w.

[5] J. Peltenburg, J. Van Straten, L. Wijtemans, L. Van Leeuwen, Z. Al-Ars, and P. Hofstee, "Fletcher: 29th International Conferenceon Field-Programmable Logic and Applications, FPL 2019," *Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019*, Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019, I. Sourdis, C.-S. Bouganis, C. Alvarez, L. A. Toledo Diaz, P. Valero, and X. Martorell, Eds., pp. 270–277, Sep. 2019, Publisher: Institute of Electrical and Electronics Engineers (IEEE). DOI: 10.1109/FPL.2019.00051. [Online]. Available: http://www.scopus.com/inward/record.url?scp=85075629579&partnerID=8YFLogxK (visited on 06/26/2022).

[6] InAccel, *FPGA deployment and scaling on distributed systems; the missing layer*, en, Mar. 2020. [Online]. Available: https://inaccel.medium.com/fpga-deployment-and-scaling-on-distributed-systems-the-missing-layer-a8b13302e789 (visited on 05/05/2022).

[7] W. David, *David Williams Is "FPGA-Curious"*, en-US, Dec. 2019. [Online]. Available: https://hackaday.com/2019/12/06/david-williams-is-fpga-curious/ (visited on 06/10/2022).

[8] fpgamarket, *FPGA Market Size, Share and Trends Forecast to 2026 | MarketsandMarkets™*. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html (visited on 05/13/2022).

[9] S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: 10.1109/JPROC.2015.2392104.

[10] xilinx, *Alveo*, en. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html (visited on 05/09/2022).

[11] *Ultrascale-plus-fpga-product-selection-guide.pdf • Viewer • Documentation Portal*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide (visited on 06/10/2022).

[12] pynqmanual, *PYNQ-Z1 Reference Manual - Digilent Reference*. [Online]. Available: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual (visited on 05/31/2022).

[13] R. Lynette, *Comparing Hardware for Artificial Intelligence: FPGAs vs. GPUs vs. ASICs – Lynnette Reese*, en-US. [Online]. Available: http://lreese.dotsenkoweb.com/2019/03/30/comparing-hardware-for-artificial-intelligence-fpgas-vs-gpus-vs-asics/ (visited on 06/10/2022).

[14] *Why the Industry is Demanding FPGAs for Advanced Driver-Assistance Systems (ADAS) - News*, en. [Online]. Available: https://www.allaboutcircuits.com/news/why-the-industry-is-demanding-fpgas-for-adas/ (visited on 05/13/2022).

[15] *Introduction to FPGA acceleration*, en. [Online]. Available: https://www.stemmer-imaging.com/en/technical-tips/introduction-to-fpga-acceleration/ (visited on 05/15/2022).

[16] J. Anderson, "High-level synthesis - the right side of history," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 1–1. DOI: 10.1109/FPT.2016.7929177.

[17] I. N. Torsvik, D. Engineer, and D. Respons, *SoC FPGA Evaluation Guidelines*, en-US, Feb. 2016. [Online]. Available: https://datarespons.com/soc-fpga-evaluation-guidelines/ (visited on 05/13/2022).

[18] *Ug573-ultrascale-memory-resources.pdf • Viewer • Documentation Portal*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources (visited on 06/10/2022).

[19] *Ds190-Zynq-7000-Overview.pdf • Viewer • Documentation Portal*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview (visited on 06/10/2022).

[20] HACC, *Heterogeneous Accelerated Compute Cluster (HACC) Program*, en. [Online]. Available: https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html (visited on 06/10/2022).

[21] *Definition of Scalability - Gartner Information Technology Glossary*, en. [Online]. Available: https://www.gartner.com/en/information-technology/glossary/scalability (visited on 06/11/2022).

[22] *Amdahl's law*, en, Page Version ID: 1089345282, May 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=1089345282 (visited on 06/10/2022).

[23] *PYNQ Libraries — Python productivity for Zynq (Pynq)*. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries.html (visited on 06/02/2022).

[24] S. Zeng, G. Dai, H. Sun, *et al.*, "Enabling Efficient and Flexible FPGA Virtualization for Deep Learning in the Cloud," en, in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Fayetteville, AR, USA: IEEE, May 2020, pp. 102–110, ISBN: 978-1-72815-803-7. DOI: 10.1109/FCCM48280.2020.00023. [Online]. Available: https://ieeexplore.ieee.org/document/9114855/ (visited on 06/11/2022).

[25] M. Quraishi, E. Bank Tavakoli, and F. Ren, *A Survey of System Architectures and Techniques for FPGA Virtualization*. Nov. 2020.

[26] *Fleet | Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. [Online]. Available: https://dl-acm-org.tudelft.idm.oclc.org/doi/10.1145/3373376.3378495 (visited on 11/10/2021).

[27] *Multiple Compute Units • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Multiple-Compute-Units (visited on 05/19/2022).

[28] M. J. Huiskes, R. E. Plessix, and W. A. Mulder, "Acoustic VTI Full-waveform Inversion with 3-D Free-surface Topography," en, *79th EAGE Conference & Exhibition 2017*, 2017, Publisher: EAGE. DOI: 10.3997/2214-4609.201700503. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Af80d9779-4905-4230-89b1-73439b9088e2 (visited on 06/11/2022).

[29] Y. Cho, R. Gibson, H. Jun, and C. Shin, "Accelerating 2-D frequency-domain full-waveform inversion via fast wave modeling using a model reduction technique," *Geophysics*, T15–T32, Jan. 2020. DOI: 10.1190/geo2018-0850.1.

[30] T. Nemeth, J. Stefani, W. Liu, R. Dimond, O. Pell, and R. Ergas, "An implementation of the acoustic wave equation on FPGAs," *Seg Technical Program Expanded Abstracts*, vol. 27, Jan. 2008. DOI: 10.1190/1.3063943.

[31] S. A. Abreo Carrillo, A. B. Ramirez, O. Reyes, D. L. Abreo-Carrillo, and H. González Alvarez, “A practical implementation of acoustic full waveform inversion on graphical processing units,” en, *CT&F - Ciencia, Tecnología y Futuro*, vol. 6, no. 2, pp. 5–16, Dec. 2015, ISSN: 0122-5383, 2382-4581. DOI: 10.29047/01225383.16. [Online]. Available: https://ctyf.journal.ecopetrol.com.co/index.php/ctyf/article/view/344 (visited on 12/08/2021).

[32] P. R. Haffinger, “Seismic Broadband Full Waveform Inversion by shot/receiver refocusing,” en, 2013. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ad2d8d264-5037-4573-8418-a079afa8d1e7 (visited on 11/25/2021).

[33] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, arXiv:1612.07119 [cs], Feb. 2017, pp. 65–74. DOI: 10.1145/3020078.3021744. [Online]. Available: http://arxiv.org/abs/1612.07119 (visited on 06/12/2022).

[34] J. Misra and I. Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” *Neurocomputing*, vol. 74, pp. 239–255, Dec. 2010. DOI: 10.1016/j.neucom.2010.03.021.

[35] *FINN*, en-US. [Online]. Available: https://xilinx.github.io/finn/ (visited on 06/02/2022).

[36] S. Balaji, *Binary Image classifier CNN using TensorFlow*, en, Aug. 2020. [Online]. Available: https://medium.com/techiepedia/binary-image-classifier-cnn-using-tensorflow-a3f5d6746697 (visited on 06/12/2022).

[37] VITISAI, *Vitis AI*, en. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html (visited on 06/02/2022).

[38] S. K. Venkataramanaiah, Y. Ma, S. Yin, *et al.*, “Automatic Compiler Based FPGA Accelerator for CNN Training,” arXiv, Tech. Rep. arXiv:1908.06724, Aug. 2019, arXiv:1908.06724 [cs, eess] type: article. [Online]. Available: http://arxiv.org/abs/1908.06724 (visited on 06/05/2022).

[39] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, ISSN: 1946-1488, Sep. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056824.

[40] XRT, *Xilinx Runtime Library (XRT)*, en. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/xrt.html (visited on 06/02/2022).

[41] *Apache Spark™ - Unified Engine for large-scale data analytics*. [Online]. Available: https://spark.apache.org/ (visited on 06/04/2022).

[42] D. developers, *State of Dask*. [Online]. Available: https://blog.dask.org/2015/05/19/State-of-Dask (visited on 05/26/2022).

[43] *PYPL PopularitY of Programming Language index*, en. [Online]. Available: https://pypl.github.io/PYPL.html (visited on 06/04/2022).

[44] *Zynq-7000 Processing System IP*, en. [Online]. Available: https://www.xilinx.com/products/intellectual-property/processing_system7.html (visited on 06/08/2022).

[45] vivado_array_partition, *Pragma HLS array_partition*, en-us, reference. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2018_2/sdaccel_doc/pragma-hls-array_partition-gle1504034361378.html (visited on 06/09/2022).

[46] *05) Array Partitioning, Reshape, and Mapping*, en. [Online]. Available: https://wikidocs.net/91000 (visited on 02/03/2022).

[47] vivado_resource, *Pragma HLS resource*, en-us, reference. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2018_2/sdaccel_doc/pragma-hls-resource-wqn1504034365505.html (visited on 06/09/2022).

[48] vivado_pipeline, *Pragma HLS pipeline*, en-us, reference. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2018_2/sdaccel_doc/pragma-hls-pipeline-fde1504034360078.html (visited on 06/09/2022).

[49] hwh, *Overlay Design — Python productivity for Zynq (Pynq)*. [Online]. Available: https://pynq.readthedocs.io/en/latest/overlay_design_methodology/overlay_design.html?highlight=hwh#overlay-hwh-file (visited on 06/10/2022).

[50] pynq_board, *PYNQ - Python productivity for Zynq*. [Online]. Available: http://www.pynq.io/board.html (visited on 06/09/2022).

[51] neumann, *Std::cyl_neumann, std::cyl_neumannf, std::cyl_neumannl - cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/numeric/special_functions/cyl_neumann (visited on 06/11/2022).

[52] bessel, *Std::cyl_bessel_j, std::cyl_bessel_jf, std::cyl_bessel_jl - cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/numeric/special_functions/cyl_bessel_j (visited on 06/11/2022).

[53] S. Aggarwal, "Scaling up data analytics in Python using multiple FPGAs," en, 2020. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ae54bbdca-3e9f-4c23-8c89-463751193061 (visited on 06/13/2022).

[54] *SSH — Dask documentation*. [Online]. Available: https://docs.dask.org/en/stable/deploying-ssh.html (visited on 06/14/2022).

[55] *Distributed*, original-date: 2015-09-13T18:42:29Z, Jun. 2022. [Online]. Available: https://github.com/dask/distributed/blob/344868ace6cd10f7b99e4cd2d3830fba66a2cdcf/distributed/deploy/ssh.py (visited on 06/14/2022).

[56] e. dask efficiency, *Efficiency — Dask.distributed 2022.6.1 documentation*. [Online]. Available: https://distributed.dask.org/en/stable/efficiency.html (visited on 06/28/2022).

[57] *Energy-efficient Cloud Computing Technologies and Policies for an Eco-friendly Cloud Market | Shaping Europe's digital future*, en. [Online]. Available: https://digital-strategy.ec.europa.eu/en/library/energy-efficient-cloud-computing-technologies-and-policies-eco-friendly-cloud-market (visited on 06/17/2022).

[58] *Global trends in internet traffic, data centres workloads and data centre energy use, 2010-2020 – Charts – Data & Statistics*, en-GB. [Online]. Available: https://www.iea.org/data-and-statistics/charts/global-trends-in-internet-traffic-data-centres-workloads-and-data-centre-energy-use-2010-2020 (visited on 06/17/2022).

[59] D. Welle (www.dw.com), *Big data centers are power-hungry, but increasingly efficient | DW | 24.01.2022*, en_GB. [Online]. Available: https://www.dw.com/en/data-centers-energy-consumption-steady-despite-big-growth-because-of-increasing-efficiency/a-60444548 (visited on 06/17/2022).