NANYANG TECHNOLOGICAL UNIVERSITY

DESIGNING HARDWARE ACCELERATORS AT HIGH LEVEL OF PROGRAMMING ABSTRACTIONS

by

RAVI PRASHANT
(G1502159C)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2016

Contents

1 Introduction **2**

1.1 Motivation 2

1.2 Contribution 3

1.3 Organization 4

2 Background **5**

2.1 Heterogenous Computing Platforms 5

2.1.1 Zedboard with a Zynq SoC 5

2.1.2 DE0-Nano-SoC with a Cyclone V SoC 6

2.1.3 The Intel CPU-GPU Platform 6

2.2 Hardware Acceleration 6

2.3 Programming Models For Hardware Accelerators 12

2.3.1 GPU 12

2.3.2 FPGA 14

2.3.3 FPGA Overlays 16

2.4 Summary 18

3 Hardware Acceleration Use Cases **19**

3.1 A Case Study On FIR Filter Execution 19

3.1.1 Effects Of Compiler Optimizations 20

3.1.2 Implementation on the VectorBlox MXP 21

3.2 A Case Study On Dynamic Loading of tasks 22

4 Experiments 25
4.1 12-Tap FIR Filter . . . 26
4.2 2D Convolution . . . 29
4.3 Kmeans . . . 32
4.4 ATAX . . . 35
4.5 BICG . . . 38
4.6 Summary . . . 41

5 Hardware Virtualization And Lab On The Cloud 43
5.1 Existing Work . . . 43
5.2 The Case for Embedded Hardware Virtualization . . . 44
5.3 The Cloud9 based virtualization for Embedded Hardware . . . 45
5.3.1 The Cloud9 IDE . . . 46
5.3.2 Node.js and the Cloud9 Server Setup . . . 47
5.3.3 The Lab On Cloud Web Application . . . 49

6 Conclusions and Future Work 51
6.1 Conclusions . . . 51
6.2 Future work . . . 53

Bibliography 54

List of Figures

2.1 Execution Platforms for Compute Kernels 7
2.2 Compute Kernel Execution on General Purpose Processor 8
2.3 Highlighting Datapath Formation in FPGA 9
2.4 High Level GPU Architecture 10
2.5 Acceleration Mechanism of Mathemetical Computations on GPU . . 11
2.6 OpenCL Flow for Computations on GPU 13
2.7 RTL Vs HLS Design Flow[1] 15
2.8 AOCL programming flow for Cyclone V Device 16
2.9 DSP block based functional unit [2]. 17
2.11 AOCL programming flow for Cyclone V Device 18

3.1 Visual Representation of the FIR filtering computation 23

4.1 OpenCL Execution Time for 1D FIR 26
4.2 OpenCL Operations Per Cycle (OPC) for 1D FIR 26
4.3 C Execution Time for 1D-FIR 28
4.4 C OPC for 1D-FIR 28
4.5 OpenCL Execution Time for 2D Convolution 29
4.6 OpenCL OPC for 2D Convolution 29
4.7 C Execution Time for 2D Convolution 30
4.8 C OPC for 2D Convolution 30
4.9 OpenCL Execution Time for Kmeans 32
4.10 OpenCL OPC for Kmeans 32

4.11 C Execution Time for Kmeans 33
4.12 C OPC for Kmeans 33
4.13 OpenCL Execution Time for ATAX 35
4.14 OpenCL OPC for ATAX 35
4.15 C Execution Time for ATAX 37
4.16 C OPC for ATAX 37
4.17 OpenCL Execution Time for BICG 38
4.18 OpenCL OPC for BICG 38
4.19 C Execution Time for BICG 39
4.20 C OPC for BICG 39

5.1 Basic Hardware Virtualization Using Cloud 45
5.2 The Zedboard-Server Setup 46
5.3 Basic Node Server Script 47
5.4 Spawning The Node Server 48
5.5 The Cloud9 IDE Webpage 49
5.6 The Cloud9 IDE project customization 50

List of Tables

3.1 T$_{task}$ in μs for Non-DMA based, Hard DMA based and ARM only implementation of FIR filters 20
3.2 T$_{task}$ in μs for Non-DMA based, Hard DMA based and ARM only implementation of FIR filters with compiler optimizations 20
3.3 Vectorblox MXP based implementation of FIR filters with compiler optimizations 22
3.4 4,8 and 12-Tap FIR Filtering on DE0-NANO-FPGA timing results with re-configuration 23
3.5 4,8 and 12-Tap FIR Filtering on Vectorblox MXP timing results with re-configuration 24

4.1 Hardware Platforms 25
4.2 Execution time of 12-Tap FIR Filter (OpenCL) on different platforms 26
4.3 OPC for 12-Tap FIR Filter (OpenCL) 27
4.4 Execution time of 12-Tap FIR Filter (C) on different platforms 28
4.5 12-Tap FIR Filter(C Implementation) OPC on different platforms . . 28
4.6 2D Convolution(OpenCL) Execution Time on different Platforms . . 29
4.7 2D Convolution(OpenCL) OPC on different platforms 30
4.8 2D Convolution(C Implementation) Execution Time on Various Platforms 31
4.9 2D Convolution(C Implementation) OPC on Various Platforms . . . 31
4.10 Kmeans(OpenCL) Execution Time on Various Platforms 32
4.11 Kmeans(OpenCL) OPC on Various Platforms 33

4.12 Kmeans (C Implementation) Execution Time on Various Platforms . 34
4.13 Kmeans (C Implementation) OPC on Various Platforms 34
4.14 ATAX(OpenCL) Execution Time on Various Platforms 36
4.15 ATAX(OpenCL) OPC on Various Platforms 36
4.16 ATAX (C Implementation) Execution Time on Various Platforms . . 37
4.17 ATAX (C Implementation) OPC on Various Platforms 37
4.18 BICG(OpenCL) Execution Time on Various Platforms 39
4.19 BICG(OpenCL) OPC on Various Platforms 39
4.20 BICG (C Implementation) Execution Time on Various Platforms . . 40
4.21 BICG (C Implementation) OPC on Various Platforms 40

Abbreviations

DSP Digital signal processing

FPGA Field Programmable Gate Array

GPP General Purpose Processor

GPU Graphics Processing Unit

HDL Hardware description language

HLS High Level Synthesis

OPC Operations Per Cycle

OpenCL Open Computing Language

RTL Register Transfer Level

Abstract

Research efforts have shown the strength of FPGA-based acceleration in a wide range of application domains where compute kernels can execute efficiently on an FPGA device. Due to the complex process of FPGA-based accelerator design, the design productivity is a major issue, restricting the effective use of these accelerators to niche disciplines involving highly skilled hardware engineers. Coarse-grained FPGA overlays, such as VectorBlox MXP and DSP block based overlays, have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction. This report presents an analysis of compute kernels (extracted from compute-intensive applications) and their implementation on multiple hardware accelerators, such as GPU, Altera OpenCL (AOCL) generated hardware accelerator and FPGA-based overlays. We experiment with simple and easy programming models like Open Computing Language (OpenCL)/Overlay APIs and produce a hardware-accelerated design with software like abstractions.

To begin we analyze two existing use-cases of hardware acceleration where one of them highlights the performance benefits obtained by use of compiler optimizations. We see that compiler optimizations can provide almost a 16× improvement in execution time on the ARM processor of the zedboard. This is because of the use of SIMD NEON engine which accelerated the execution. The use of MXP overlay for the same application provides an even higher improvement in the execution time compared to SIMD NEON engine. The other hardware acceleration case study analyses the feasibility of dynamic loading of tasks to the FPGA fabric and the effect on the execution time. We use AOCL to create accelerators for multiple tasks and then using the software API, we perform the dynamic reconfiguration. We show that the use of overlay is preferable in such a scenario due to their ease of use, simple programming model and dynamic task loading without actual reconfiguration of the FPGA fabric. When

the same task was executed on an overlay it ran much faster as there is no need to reconfigure the FPGA fabric with a new bitstream.

We present experiments to compare the performance of a naive implementation of few compute kernels with their hardware accelerated versions that were built either using OpenCL or using Overlay APIs. We observe up to 10$\times$ improvement in timing performance in certain applications like 12-Tap FIR filtering when accelerated using hardware and almost 100$\times$ in certain applications like 2D Convolution. These performance improvements were obtained by using very basic and naive implementation of hardware accelerators generated at a high level of programming abstractions (OpenCL/Overlay APIs). With optimizations, the performance can surely be improved, this would be one of the key areas of future research work beyond this thesis. Finally, we make the case for hardware virtualization by using the cloud and demonstrate how by means of a simple web browser we can program remote computing platforms connected to the cloud servers. Such virtualization methods could be used in teaching labs and for hardware evaluations.

Acknowledgment

First and foremost, I would like to thank Assoc Prof Dr. Douglas Leslie Maskell for his guidance, enthusiastic support, and strong encouragement. I would not have been able to complete my project successfully without his support and directions.

Moreover, I would also like to thank Abhishek Kumar Jain for his professional guidance, continuous support, effective suggestions, constructive criticism and timely help. I am also thankful to him for carefully reading and commenting on countless revisions of this report.

Thanks to Mr. Jeremiah Chua in the Hardware And Embedded Systems Lab (HESL) for his technical support and the facilities.

Finally, I am indebted to my family and my parents for their prayers and encouragement. I thank them for their understanding and their efforts to support me in pursuing higher studies.

Chapter 1

Introduction

1.1 Motivation

The Moore's law has almost come to an end and not much performance up gradation can be achieved just by upgrading hardware generations. The methods of frequency scaling and core multiplication have reached their maximum limits of exploitation and cannot be explored further. This gives rise to the need for heterogeneous computing. Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks.

There are a number of heterogeneous devices that are readily available off the shelf, some of them as commercially available system on chips (SoCs) and can be easily accessible to the average embedded software programmer. One of the example is Xilinx Zynq. These heterogeneous devices incorporate hardware accelerators, such as GPUs and FPGAs. The advantages of having multiple types of hardware accelerators within a system on chip are many, such as high-performance gains and specificity of tasks performed. One of the major issues is the exotic programming model that each hardware accelerator brings along with it.

FPGAs bring along the requirement to write accelerators using hardware description language (HDL) at a very low level of abstraction (register transfer level) which

has a really steep learning curve, extremely long compilation/place and route timings and are relatively difficult to debug. GPUs have their own programming languages and until a few years ago GPUs were used predominantly for graphics processing as the name suggests. However off late, with computing problems of large sizes and datasets, a lot of computations are being off-loaded to the GPU due to its benefit in parallel execution. Modern GPUs support programming models like OpenCL and CUDA.

Even with a great number of devices available, the difference in programming models makes it difficult to truly use all these devices in unison. Even with these multitudes of methods available for hardware acceleration, it is still not possible for software developers to adopt and use these devices as part of their daily routine for hardware acceleration. In order to achieve the full potential of hardware acceleration, multiple heterogeneous devices within the same SoC must be programmable using a single programming model and also the programming model must be portable across devices without loss of functionality. The aim of this thesis is to investigate and benchmark novel heterogeneous devices and methods by which we can use heterogeneous devices with a fairly limited learning curve.

1.2 Contribution

The main contributions can be summarized as follows:

- Proposed an approach for performing hardware acceleration by means of software based high-level design.
- Developing an understanding of compute kernels and their benchmarking with hardware acceleration on overlay and GPU architectures with detailed performance evaluation of these devices.
- Performance comparison of different platforms and analysis of the effect of hardware acceleration in each case.
- Proposal and proof of concept of hardware virtualization by means of a cloud server with an online lab on a cloud programming model.

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on computer kernels, overlay architectures and various accelerators that have been studied. In chapter 3.1 and 3.2, we perform case studies using the 12-Tap FIR filter as the benchmark and port the algorithm to the DSP overlay, Vectorblox MXP, and Altera Cyclone V device using AOCL. In Chapter 4, we perform a number of experimental benchmarks on various platforms with hardware acceleration and have compared multiple programming models. The performance of the hardware platforms in terms of execution time and Operations per Cycle (OPC) has been measured and compared highlighting the ease of programming with hardware acceleration. Chapter 5 illustrates how virtualization of embedded hardware can be done and implemented on a cloud-based platform.

Chapter 2

Background

2.1 Heterogenous Computing Platforms

A heterogeneous computing platform is one which has multiple processing units instead of one single processing unit on the same SoC. The multiple processing units are designed such that each unit is tuned to perform a specific task, such as Signal processing, graphics processing. The most common element found in any embedded hardware platform is a CPU (Central Processing Unit) which is generally a General Purpose Processor (GPP) and the other co-processing units that are used are GPUs, FPGAs DSPs etc. The heterogeneous computing devices that are going to be extensively used in this thesis are the following:

2.1.1 Zedboard with a Zynq SoC

The Zedboard is an FPGA based heterogeneous platform that is made by avnet[3] and has a Xilinx Zynq SoC[3]. The SoC has a dual core ARM processor clocked at 667 MHz and also has a programmable logic which is connected to the CPU through the AXI bus[3]. The Zedboard is pretty widely used in academia as well as industry and is a proven platform. Some of the wide applications where the Zynq SoC has been used ranges from Software defined Radios[4] and extends even to data processing applications in space satellites[4]. This wide range of applications has made the

Zedboard an ideal candidate for heterogeneous computing experiments in this thesis.

2.1.2 DE0-Nano-SoC with a Cyclone V SoC

The DE0-Nano-SoC is another heterogeneous computing platform that is similar to the Zedboard. The DE0-Nano-SoC board has an Altera Cyclone V SoC. This SoC has a dual core ARM processor clocked at 995 MHz and has a programmable logic fabric as well. The DE0-NANO stands out from the Zedboard due to its compatibility with all Arduino Shield boards, This makes it a more popular choice amongst hobbyists and makers since hardware can be easily interfaced to the DE0-NANO also, the board ships with a custom open-source Arduino hardware design built into the FPGA which gives it full Arduino capability and possibility of functionality scaling. Also, the Cyclone V supports Altera's OpenCL SDK which makes it easier to use by the regular software programmer.

2.1.3 The Intel CPU-GPU Platform

In order to demonstrate code portability across multiple devices, the conventional Intel CPU-GPU compute platform has also been considered. The Intel CPU used in this thesis is the Intel Core i7 clocked at 1.6 GHz and the Intel iris 6100 GPU clocked at 1.1 GHz. These devices are certainly not within the embedded domain but are part of this thesis for the purposes of comparison and demonstration of code portability and ease of hardware acceleration in conventional hardware devices as well.

2.2 Hardware Acceleration

Hardware acceleration is a process by which the GPP on an embedded hardware platform off-loads specialized types of computations to specialized cores that are optimized for doing that specific task. This method of off-loading specialized tasks to specific units on the same SoC leads to faster execution of the tasks and also allows the CPU to proceed with any other task in the mean time thus giving rise to task level parallelism.

A GPP can be used for the execution of compute kernels by describing their functionality using C or C-like programming languages. With the advancements in technology, parallel processing architectures such as multi-cores CPUs and Digital signal processing (DSP)s, GPUs, Massively parallel processor arrays, FPGA-based accelerators (as shown in Fig. 2.1) are gaining popularity for accelerated execution of kernels.

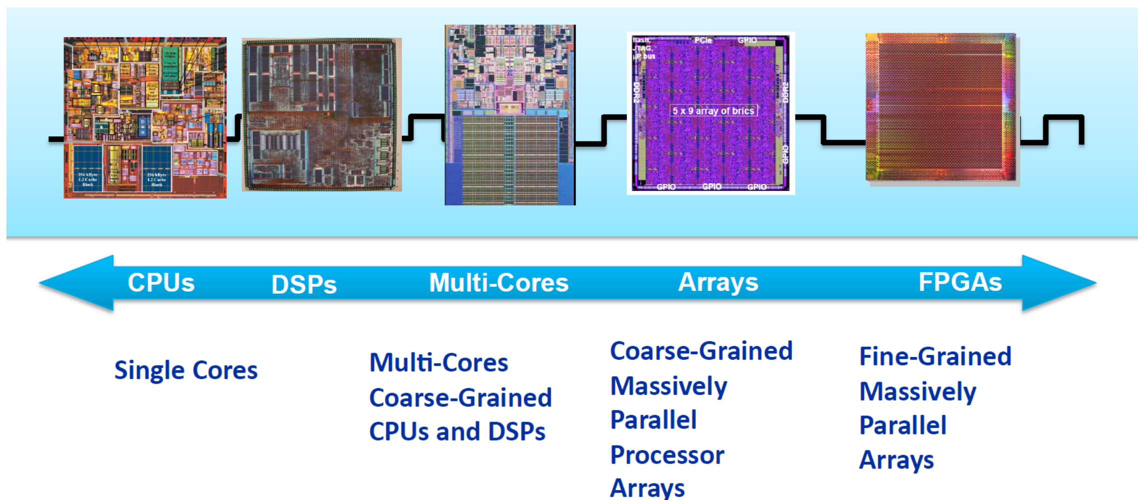


Figure 2.1: Execution Platforms for Compute Kernels

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional processor designs will not be able to meet. On the other hand, Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) devices provide a substrate for implementing kernels as high performance fully parallel and pipelined designs [5]. Just to provide the clear understanding of the concept, we explain the execution of compute kernels on general purpose processors, FPGAs and GPUs in this thesis.

As mentioned earlier, a general purpose processor can be used for the execution of compute kernels by describing their functionality using C or C++ like programming languages. A compiler then generates a list of instructions for the processor to execute sequentially. Since the processor executes the list of operations sequentially, the execution time of the kernels increases on increasing the complexity of the kernel. Fig.

2.2 shows the high-level view of program execution on a general purpose processor.

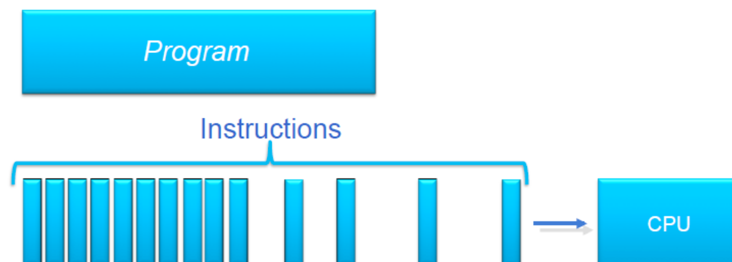


Figure 2.2: Compute Kernel Execution on General Purpose Processor

Having an extremely powerful GPP might not always be advantageous. Specialized tasks might need highly specialized compute units in order to perform the task efficiently. For example signal processing tasks are best done by a DSP, graphics processing is best done when offloaded to a GPU. Also by keeping on making the CPU more and more powerful, silicon manufacturers have reached the peak of voltage scaling as well as frequency scaling[6][7]. This effectively makes the case for hardware accelerators like FPGAs and GPUs to come into the picture.

The case for FPGA-based heterogeneous computing: FPGAs are becoming popular for rapid-prototyping of accelerators. For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of software, in some cases by several orders of magnitude compared to state-of-the-art general purpose processors [8, 9]. To understand the execution of kernels on FPGAs, we must first understand how FPGA architectures differ from general purpose processor architectures. The most fundamental difference is that general-purpose processors provide functionality to execute a list of instructions sequentially, whereas FPGA architectures implement compute kernels by providing numerous resources such as configurable logic blocks, DSP blocks for logic and arithmetic and on-chip Block RAMs for storage. These resources are generally interconnected via a programmable island-style routing network which can be programmed to create specialized data paths as shown in Fig. 2.3.

Also as mentioned in [6] the most efficient solution to the frequency scaling problem, is the creation of customized data paths which can be efficiently done on an

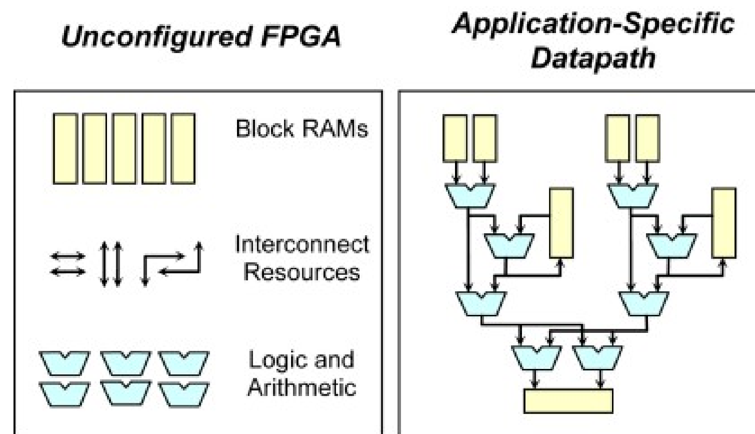


Figure 2.3: Highlighting Datapath Formation in FPGA

FPGA. FPGA accelerators are normally designed at a low level of abstraction (typically Register Transfer Level (RTL)) in order to obtain an efficient implementation, and this can consume more time and make reuse difficult when compared with a similar software design. As such, design productivity remains a major challenge, restricting the effective use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers. In order to tackle issues with design productivity, we highlight certain key methods in this thesis which allow even novice engineers to make use of FPGA resources. The three methods by which we demonstrate programming FPGAs are :-

- High-Level Synthesis
- OpenCL To Hardware Design
- FPGA Overlays

These methods are described and their efficiency is demonstrated in the experiments section in chapter 4.

The case for GPU-based heterogeneous computing: GPUs have been there in SoCs for a long time and have been extensively used in high-end graphics and gaming systems. With the improvement in fabrication technologies, GPUs have become only smaller and much more powerful. GPUs have become an integral part of

modern SoC's right from high-end server racks to tiny mobile devices, we can find a GPU on almost every SoC. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor which is fully capable of performing massively parallel mathematical and statistical problems [10]. Due to its wide usage nowadays in non-graphics applications, a new term *GPGPU* has been coined which stands for General Purpose GPU. The architecture of a GPU is shown in figure 2.4

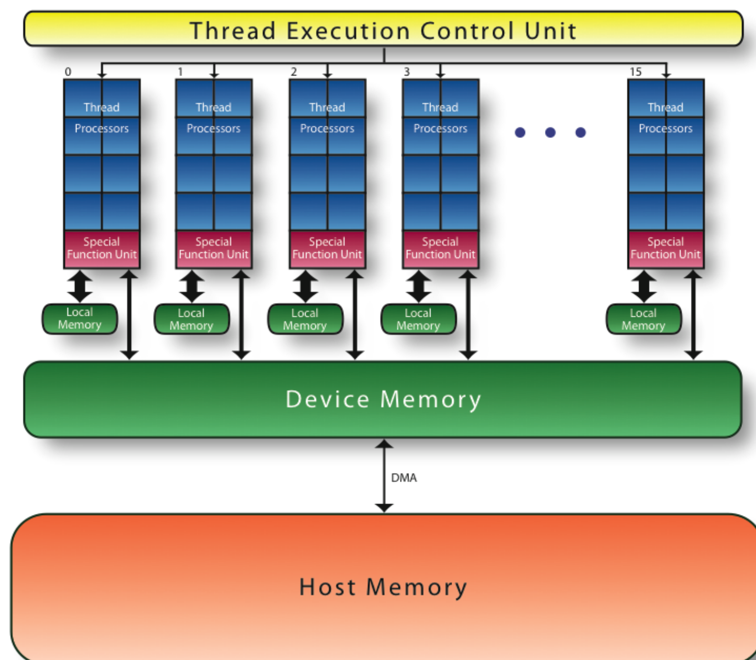


Figure 2.4: High Level GPU Architecture

As shown in the figure 2.4 the GPU has its own device memory and a set of thread processors. Essentially when a computation is offloaded to a GPU, it is divided into a number of threads and executed in parallel in the thread processors shown. This parallelism results in an overall speed-up of the task that has been off-loaded to the GPU. This type of GPU architecture is extremely useful when we have an extremely large number of computations that are of the same type or operation without any data dependency with each other for example matrix multiplication of large matrices.

In large matrix multiplications, it is only a repeated process of addition and multiplication on a large set of numbers and many of these computations can be done entirely in parallel, on a CPU each computation in matrix multiplication must wait for the previous computation to finish before it gets executed. The example of how GPU helps in parallel execution of computations is shown in figure 2.5.

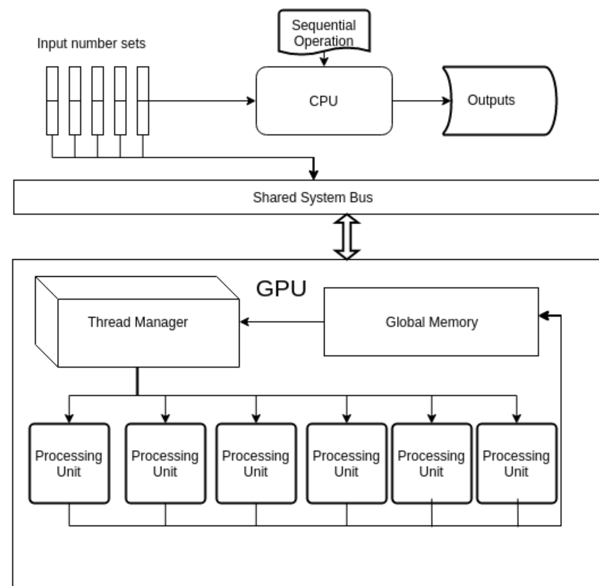


Figure 2.5: Acceleration Mechanism of Mathemetical Computations on GPU

The basic difference between a GPU and an FPGA is that the architecture inside the GPU is always fixed and will have a similar kind of programming model and data flow all through its lifetime. FPGA, on the other hand, gives the user flexibility to modify and change the hardware design inside the SoC as per application in order to extract maximum efficiency from the customized data flow path.

2.3 Programming Models For Hardware Accelerators

There are a number of programming models that are used for hardware acceleration with different devices. In the case of FPGAs, the most common programming languages are Hardware description language (HDL) eg:- Verilog and VHDL. To use an FPGA for accelerating compute kernels, designers typically start by manually converting the compute kernel into a fully pipelined data path as shown in Figure 2.3, specified using HDL. A fully pipelined datapath on FPGA results in maximum performance by producing output data at every clock cycle. However, this performance comes at the cost of designer effort. The various design challenges faced by this type of FPGA programming model is listed:

- High learning curve for HDL
- Large compilation times
- Code for large designs becomes very complex to read and manage

In order to tackle these problems of HDL design, we have analyzed alternate programming models that could also be effectively used for hardware acceleration with much shorter learning curves and more importantly without the use of HDL. It is to be noted here that this thesis does not discourage the use and benefits of HDL. The analyses were done and methods experimented with are ones that use a form of high level translation/synthesis of traditional software code to hardware designs. FPGA designs using HDL are out of the scope of this thesis.

2.3.1 GPU

As mentioned in subsection 2.2 GPGPUs are very common in most modern SoCs. The specialty of GPGPUs is that they are general purpose in nature and can be programmed with a common programming language that can be used to program multiple applications onto the GPU. The two most common programming languages

for the GPU are OpenCL and CUDA [11]. The CUDA programming was made for programming NVIDIA GPUs and the language can only be used to program NVIDIA GPUs and hence its application is pretty limited. On the other hand, OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators [12].

This thesis shall focus on using OpenCL to program the Intel Iris 6100 GPU which generally ships with many common Intel CPU's inside the same SoC. This thesis focuses more on the OpenCL C API. The C API for OpenCL is maintained by Khronos group[13]. In OpenCL programming model, there are two parts of the code namely the host and the kernel. The host code is written in C while the kernel is a C-like representation of the computation that needs to be offloaded onto the GPU. The host code and kernel code are compiled separately. The host code is compiled to the CPU while the kernel is compiled for the GPU. The host is responsible for transferring the data to be processed upon to the device memory and read back the results from the device memory and triggering the device to start executing the kernel. Figure 2.6 shows the high-level flow of how OpenCL can be used to accelerate kernels on a GPU.

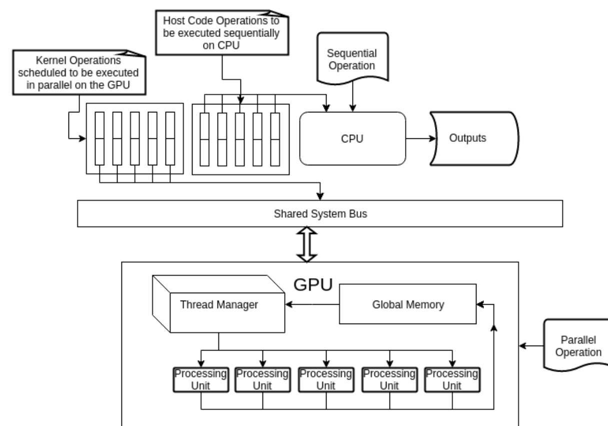


Figure 2.6: OpenCL Flow for Computations on GPU

2.3.2 FPGA

The methods of programming an FPGA without the use of HDL are many but in this thesis, we shall discuss mainly three methods of hardware acceleration sans the use of HDL. Though HDL may be the most efficient way of programming an FPGA but alternate methods that will be discussed further in this section will enable almost all novice programmers write code and run it on an FPGA. The three alternate methods of programming FPGA's mentioned in 2.2 are detailed below.

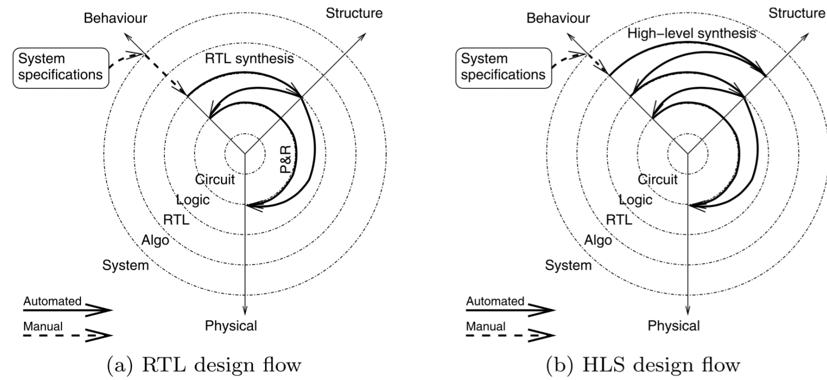
High-Level Synthesis:

High-level synthesis (High Level Synthesis (HLS)) is an increasingly popular approach that is being used nowadays to generate digital circuits to be instantiated on FPGA's[1]. When HDL is used the designer must translate the designs from the algorithmic level all the way up to the RTL level. With the use of high-level synthesis, this step is abstracted to a few steps above the RTL level. To illustrate this difference between the RTL flow and the HLS flow has been illustrated by Wim Meeus et. al. in [1] using the Gasjki-Kuhn Y-chart[14]. In the Y-Chart for the RTL flow, the designer manually converts the high-level system specifications all the way to the RTL level after which automatic place and route of the RTL takes place. In the case of HLS flow, the designer converts the system specification into an algorithmic representation usually in C/C++ or any high-level programming language which is then automatically translated from the algorithmic level all the way to the actual hardware generation after place and route. This concept has been highlighted by figure 2.7.

Some of the popular tools for High-Level Synthesis are Xilinx Vivado HLS and Xilinx SDSoC. Even though HLS is a very compelling tool, this thesis shall not evaluate HLS but shall look into a subset of HLS that is explained in the next section.

OpenCL:

Due to the increasing popularity of OpenCL as a programming language, a number of FPGA vendors have shown interest towards this by developing OpenCL SDK for FPGA-based SoCs. Two of the most famous OpenCL SDKs for FPGAs are the SDAccel tool built by Xilinx for their PCIe based FPGAs [15], and the Altera OpenCL(AOCL) SDK developed by Altera primarily for its Cyclone V devices [16]. This thesis focuses more on analyzing and benchmarking of AOCL on Cyclone V



(a) RTL design flow (b) HLS design flow

Figure 2.7: RTL Vs HLS Design Flow[1]

devices as this fits more the embedded context.

In this thesis, specifically, the DE0-NANO-SOC board has been used which houses a Cyclone V SoC. As explained in subsection 2.3.1 the OpenCL programming language consists of a host code which manages thread launch on the GPU and data transfer to and from the GPU while the kernel is the actual computation workload that is accelerated. Using the same principle the AOCL uses a host code that runs on the CPU of the SoC which is responsible for data transfers/DMA's and execution control while the kernel is translated into hardware design using a similar HLS approach as explained in figure 2.7. AOCL, in particular, gives a lot of scope for tweaking and modifying the kernel in order to place and route efficiently to the FPGA, for example in the kernel we can use *pragmas* to drive the optimization in the way we want. For example, we could create more than one hardware implementations of the same kernel within the same FPGA and enable parallel processing. The AOCL SDK has been used with the DE0-NANO-SOC and a number of benchmarks and applications have been tested, results of which are detailed in the experiments section4. The AOCL OpenCL flow is almost similar to the OpenCL Flow for GPUs with the following caveats. The kernel is compiled beforehand using AOCL SDK. Secondly, the kernel is compiled offline and the bitstream to program the FPGA is generated. Figure 2.8 shows the AOCL programming flow for the Cyclone V SoC.

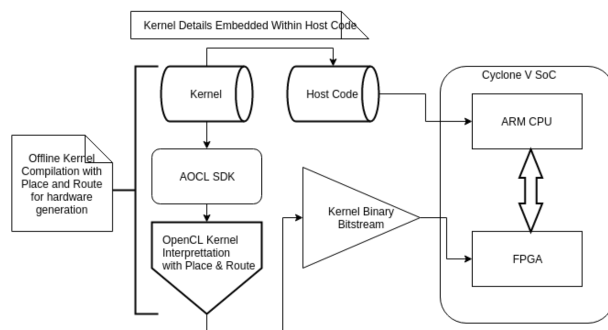


Figure 2.8: AOCL programming flow for Cyclone V Device

2.3.3 FPGA Overlays

Overlay architectures consist of a regular arrangement of coarse-grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through the mapping from high-level descriptions, application portability across devices, design reuse, a fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Another main advantage is rapid reconfiguration since the overlay architectures have smaller configuration data size due to the coarse granularity. Accelerators can be described at a higher level of abstraction and compiling it for overlays is several orders of magnitude faster than for the fine-grained FPGAs. Researchers have proposed fine [17], [18] and coarse grained [19], [20], [21], [22], [23], [24] overlay architectures to abstract FPGA fabric resources. Coarse-grained configurable overlay architectures have been proposed as a method to overcome some of these issues [19, 20, 21, 22, 25, 23]. Overlays can be used for reducing the prohibitive compilation time required to map an application to the conventional fine-grained FPGA fabric. Overlays have also been shown to be effective when paired with general purpose processors [26, 20] as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use.

Two overlays have been studied in this thesis, one of them is an FU-based DSP overlay made with DSP blocks as proposed in [2]. The other overlay is a more popular one known as MXP made by Vectorblox[27]. The Vectorblox overlay is a soft vector processor with 16 Vector lanes and this was instantiated on the FPGA fabric of the Zedboard. Detailed descriptions of these overlays are below:

DSP Overlay: An overlay architecture with the Functional Unit(FU) based on the DSP blocks found in Xilinx FPGAs was recently proposed [2]. This overlay combines multiple operations in a compute kernel and maps them to the DSP block, resulting in a significant reduction in the number of processing nodes required. An F_{max} of 370 MHz with throughputs better than that achieved by directly implementing the benchmarks onto the fabric using Xilinx Vivado HLS were reported.

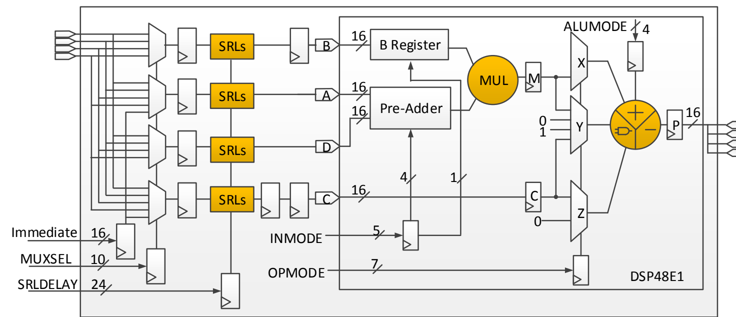
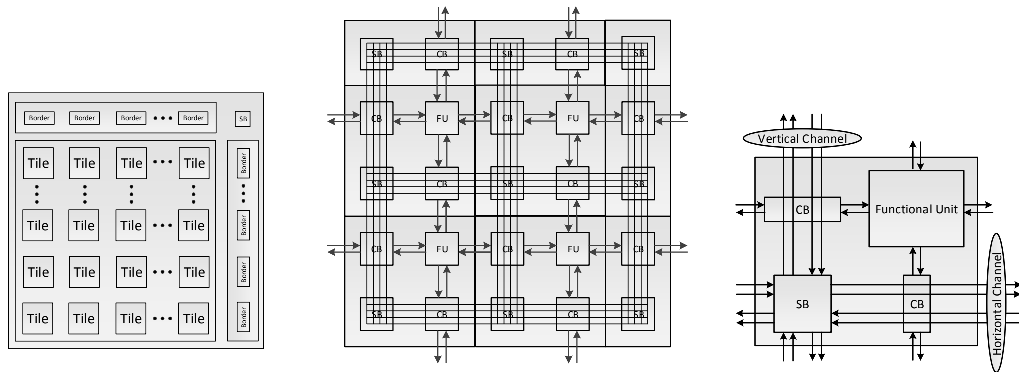


Figure 2.9: DSP block based functional unit [2].



(a) Overlay block diagram. (b) Architecture of a 2×2 Overlay. (c) Tile architecture.

Figure 2.10: DSP Block based architecture as Island-style overlay.

This DSP overlay has been used to accelerate FIR filtering and has been detailed in chapter 3.1.

Vectorblox MXP Overlay: The VectorBlox MXP Matrix Processor[27] is an FPGA-based soft processor that supports parallel execution of computations. It can

be programmed entirely in C and C++[27]. The MXP processor is built to handle data-parallel software algorithms at hardware-like speeds. MXP's parameterized design lets the user specify the amount of parallelism required, ranging from 1 to 128 or more parallel ALUs. Key features of the MXP include a parallel-access scratchpad memory to hold vector data and a high-throughput DMA engine. It excels at applications like image processing where every pixel in a large set of data is subjected to the same sequence of calculations. With just a single instruction, the MXP can apply an operation to a vector, matrix, submatrix, volume, or sub-volume of data. The architecture of the mxp processor instantiated on the Zedboard is shown in figure 2.11.

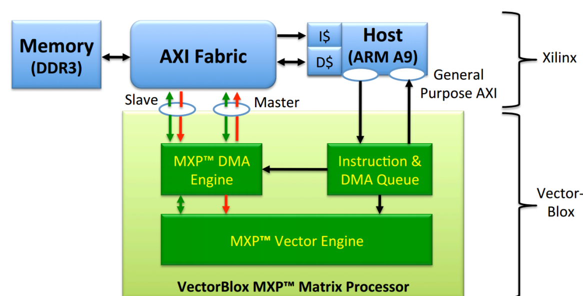


Figure 2.11: AOCL programming flow for Cyclone V Device

The MXP, in particular, has been compared with both the DE0-NANO and the DSP overlay in two case studies that have been detailed in chapters 3.1 and 3.2.

2.4 Summary

This chapter touches upon at a high level all the hardware platforms that will be covered in this thesis and details some of the advantages and disadvantages of the same. The next two chapters deal with a few of the case studies performed on some of the devices and comparisons with existing work. The experiments section discusses and analyses a number of applications and benchmarks run on the various hardware platforms and their performance benchmarks.

Chapter 3

Hardware Acceleration Use Cases

3.1 A Case Study On FIR Filter Execution

This chapter presents a case for hardware acceleration using bare metal execution on embedded devices, specifically on Xilinx Zynq. To examine the communication overheads, we use three simple 4, 8 and 12 tap FIR filter implementations which we have an implementation on ARM processor, MXP overlay and DSP overlay [26]. The original implementation of the filter is taken from [26] and the naive un-optimized results are shown in Table 3.1. T_{task} shows the time in μs which is the sum of data communication time (to and from the accelerator) and the data processing time. For each filter, there are three different implementations, first uses ARM processor for data processing, second uses DSP overlay for data processing where ARM processor is involved in data transfer from external memory to the overlay (communication bandwidth only 25 Mbytes/second), third also uses DSP overlay where hard DMA is involved in data transfer to provide higher communication bandwidth (a maximum of 80 Mbytes/second). More details are given in the experiment section of [26]. These small examples were chosen as they better demonstrate the overheads of the hybrid system, without task execution time dominating. It is possible to expand these tasks to more practical, larger ones with minimal effort. The operating frequency of the ARM processor, MXP overlay, and the DSP overlay is 667 MHz, 100 MHz, and 100 MHz, respectively.

Table 3.1: T_{task} in μs for Non-DMA based, Hard DMA based and ARM only implementation of FIR filters

Number of Samples	4-tap			8-tap			12-tap		
	ARM	Non DMA	Hard DMA	ARM	Non DMA	Hard DMA	ARM	Non DMA	Hard DMA
64	19.62	17.05	24.40	36.73	20.10	27.56	55.380	23.87	30.72
128	38.98	29.53	28.55	73.01	32.26	31.71	109.92	36.23	34.87
256	77.41	54.50	37.58	145.67	56.57	40.74	219.32	61.18	43.9
512	153.32	101.85	55.56	292.83	105.19	58.72	438.27	108.69	61.88
1024	307.56	198.57	89.33	587.27	199.89	92.49	872.10	205.40	95.65

As seen in Table 3.1, the DSP overlay implementation of the FIR filter performs much better than the ARM processor on the Zedboard in the case when hard DMA is involved in the data communication. For example, in the case of 12 tap FIR filter, a speed up of $\approx 10\times$ was achieved when 1024 samples are processed. It is to be noted here that these results have been obtained without the use of any compiler optimizations.

3.1.1 Effects Of Compiler Optimizations

The same implementations were taken for the 4,8 and 12-Tap FIR filtering and was accelerated using compiler optimizations. This was done to analyze how well the compiler optimizations improve the overall performance of the filter algorithm. The compiler optimizations were used such that auto-vectorization of the existing code was enabled in order to make use of the SIMD NEON vector engine in the Zynq device. After performing compiler optimization, the results are obtained and shown in table 3.2.

Table 3.2: T_{task} in μs for Non-DMA based, Hard DMA based and ARM only implementation of FIR filters with compiler optimizations

Number of Samples	4-tap				8-tap				12-tap			
	ARM BM	ARM LIN	Non DMA	HARD DMA	ARM BM	ARM LIN	Non DMA	HARD DMA	ARM BM	ARM LIN	Non DMA	HARD DMA
64	1	1.3	15	14	2	2.4	18	17	3	4.5	21	21
128	3	3.5	25	16	5	5.8	28	19	7	9	31	23
256	6	5	46	20	8	9.6	49	23	14	17	52	27
512	10	10	88	31	18	19	91	34	28	37	94	37
1024	21	17	152	48	28	33.5	155	51	53	66	158	53

As we can see from Table 3.2 the compiler optimization allows fast execution. In the first case, where ARM processor is used for data processing, we observe $\approx 16\times$ speed-up. This is due to the effective use of SIMD NEON engine for vector processing which outperforms the DSP overlay implementation in almost every case. It can be inferred from this experiment that hardware acceleration as proposed in [26] becomes ineffective when the number of samples to be processed are very less (less than 1K samples). Although DSP overlay (given a high-speed communication interface around the overlay) can provide a performance of 700 MOPS, 1500 MOPS and 2300 MOPS in the case of 4, 8 and 12-tap filter, respectively, the performance of the design in [26] is limited by the use of low performance general purpose (GP) port for data communication. We observe a performance of up to 550 MOPS for NEON SIMD engine which is calculated based on the fact that 1K samples are processed using a 12-tap filter (12 multiplication and 11 addition operations per sample) in 53 μs, in other words, 23K operations are performed in 53 μs. We believe that a high-speed communication interface around DSP overlay would be able to give us $4\times$ performance speed-up compared to SIMD NEON engine.

3.1.2 Implementation on the VectorBlox MXP

The VectorBlox MXP overlay[27] is a vector processor implemented on top of FPGA fabric. It uses multiple ALU lanes to process data streams in an SIMD fashion. The most important part is the communication interface and scratchpad memory around multiple ALU lanes which is optimized heavily to provide high bandwidth ($\approx$ 800 Mbytes/sec) data communication between external memory and MXP overlay execution units. It uses high performance (HP) ports of Zynq device for data communication between external memory and the overlay. Theoretically, in each cycle, 8 Bytes of data can be transferred through the HP port. In order to explore the performance benefits we implement the filtering benchmarks on the MXP overlay. Since the MXP overlay that we use consists of 16 ALUs and runs at 100 MHz, the maximum performance should be limited to 1600 MOPS due to which we expect MXP to outperform SIMD NEON engine. We FIR filters on the MXP overlay which

runs on top of the FPGA fabric. The results are shown below in Table 3.3.

Table 3.3: Vectorblox MXP based implementation of FIR filters with compiler optimizations

Samples	**4-Tap**	**8-Tap**	**12-Tap**
64	3	4	5
128	6	6	7
256	12	12	13
512	24	23	24
1024	41	41	45

As seen from table 3.3, in the case of the 12-tap filter, the MXP overlay performs 20% better than the SIMD NEON engine. The small gain in the performance is due to the use of a small number of data samples. MXP overlay generally performs well when the number of data samples to be processed is high. The case-study in this chapter shows that FPGA-based overlays (MXP and DSP overlay) have the potential to outperform hard vector processing engine (SIMD NEON). However, the performance benefits would be satisfactory when processing a large amount of data samples including a high-speed communication interface.

3.2 A Case Study On Dynamic Loading of tasks

One of the major benefits of FPGA as a rapidly reconfigurable hardware accelerator in a heterogeneous computing platform is to utilize the ability to dynamically reconfigure the functionality of the FPGA fabric. We consider Altera SoC device (Cyclone V on DE0-Nano-SoC kit) as a heterogeneous computing platform in this chapter to explore the feasibility of dynamic loading of tasks to FPGA fabric.

The Cyclone V SoC on DE0-Nano-SoC kit has the capability of being reconfigured on the fly. This method of runtime reconfiguration opens up the possibilities of dynamically loading new tasks in order to solve different parts of a large application. This especially is advantageous when the accelerator designed for a large application does not fit on the FPGA fabric. The bit-streams for multiple tasks can be stored in

a persistent storage medium like an SD Card or an internal flash memory and can be swapped onto the fabric when necessary.

In this chapter, we perform a small case study on the Cycle V FPGA device to test the capabilities of runtime reconfiguration of the FPGA. The application chosen was FIR filtering where the input samples were first run through a 4-Tap FIR filter and then through an 8-Tap FIR filter and finally through a 12-Tap FIR filter. A visual representation of the computation is shown in figure 3.1. The assumption here is that all of the filters can not fit simultaneously onto the FPGA fabric and due to which we have to perform runtime reconfiguration.

Figure 3.1: Visual Representation of the FIR filtering computation

The computation is shown in the Fig 3.1 was implemented using Altera OpenCL (AOCL) SDK on a DE0-Nano-SOC kit. The three kernels for each phase of the FIR filtering are translated into hardware designs using the AOCL SDK. The run-times for execution as well as reconfiguration of the FPGA are as shown in table 3.4.

Table 3.4: 4,8 and 12-Tap FIR Filtering on DE0-NANO-FPGA timing results with re-configuration

Samples	**4-Tap**	**8-Tap**	**12-Tap**	**1st Re-config**	**2nd Re-config**
64	80	232581	231425	376166	349106
128	81	232955	234140	375742	376857
256	85	234432	234516	375938	349158
512	86	234455	235146	376001	375311
1024	94	262807	271443	375922	348984

It is clear from the table that significantly large time is taken for the reconfiguration of the FPGA. As a next step, we perform the same experiment using an overlay, in this case, MXP overlay. The results are shown in Table 3.5.

Table 3.5: 4,8 and 12-Tap FIR Filtering on Vectorblox MXP timing results with re-configuration

Samples	**4-Tap**	**8-Tap**	**12-Tap**
64	2.689	2.721	2.76
128	5.25	5.27	5.37
256	10.36	10.4	10.44
512	20.6	20.64	20.67
1024	41.08	41.12	41.16

It is clear from the Table 3.5 that the vectorblox MXP overlay is much better at handling this type of workloads with ease as there is no need of actual reconfiguration of the FPGA. This chapter concludes that the overlay architectures can be preferred due to their ease of use, simple programming model and dynamic task loading without actual reconfiguration of the FPGA fabric.

Chapter 4

Experiments

In this chapter, we present experiments to evaluate the performance of various benchmarks (in terms of computation time and operations per cycle(OPC)). The data presented here would help judge as to how each application performs when run on different platforms under different acceleration criteria. In order to perform benchmarking, the following applications were identified and chosen for comparison.

- 12-Tap FIR Filter
- 2D Convolution
- Kmeans
- atax
- bicg

Table 4.1: Hardware Platforms

CPU	**GPU**	**FPGA**
Intel Core i7	Intel Iris 6100	Terasic DE0-NANO-SOC (Altera Cyclone V)
Avnet Zedboard (Arm v7)		Avnet Zedboard (Zynq 7000)
Terasic DE0-NANO-SOC (Cortex A9)		

The first three applications have been developed by us while the last two have been taken from a standard benchmark suite known as polybench [28]. Table 4.1 shows the hardware platforms used for benchmarking.

4.1 12-Tap FIR Filter

OpenCL Implementation:

The 12-Tap FIR filter was implemented in OpenCL and the following graphs show the performance gaps between the various platforms. Table 4.2 shows the execution time for different number of samples.

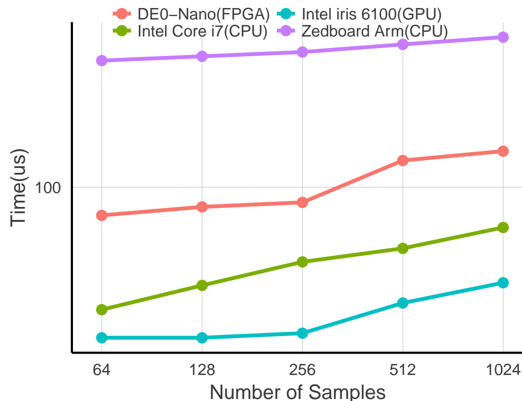


Figure 4.1: OpenCL Execution Time for 1D FIR

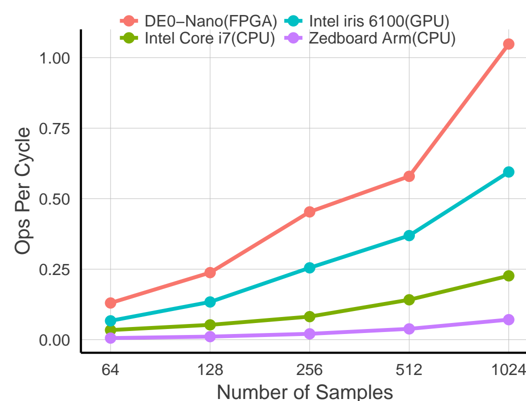


Figure 4.2: OpenCL OPC for 1D FIR

Table 4.2: Execution time of 12-Tap FIR Filter (OpenCL) on different platforms

Samples	**Intel Core i7**	**Intel Iris 6100**	**Zedboard ARM v7**	**DE0-NANO FPGA**
64	27	20	387	74
128	35	20	405	81
256	45	21	424	85
512	52	29	460	133
1024	65	36	497	147

Fig. 4.1 shows the timing gaps between the different platforms shown. It can be clearly seen that the GPU is leading with the least execution time due to the parallelism involved and the execution time is very high in case of Zedboard ARM processor. Use of AOCL generated hardware tries to improve the performance of

Table 4.3: OPC for 12-Tap FIR Filter (OpenCL)

Samples	**Intel Core i7**	**Intel Iris 6100**	**Zedboard ARM v7**	**DE0-NANO FPGA**
64	0.034	0.0669	0.00570	0.1301
128	0.0525	0.1338	0.01089	0.2378
256	0.08177	0.25489	0.0208	0.4532
512	0.1415	0.3691	0.0383	0.5793
1024	0.2264	0.59474	0.0710	1.0483

Altera Cyclone V ARM processor by offloading the OpenCL kernel execution on the accelerator. However, Intel Core i7 and Intel Iris 6100 GPU always outperform embedded platform (Cyclone V ARM processor with AOCL generated accelerator). This is not surprising as the operating frequency of Intel CPU and GPU is way too high compared to embedded platform.

If we look at operations per cycle(OPC) in Figure 4.2, the embedded platform outperforms others by crunching much more operations as compared to the other platforms in a fixed period of time. This could be attributed to the highly efficient hardware design of the OpenCL kernel offloaded to the FPGA fabric.

C implementation: When the same application was implemented in ANSI C and was run on the hardware platforms, the results obtained are as shown in figure 4.3 and 4.4. In the ANSI C benchmarking we have introduced the Vectorblox MXP overlay as well as it can be easily programmed using a set of C APIs. The timing performance clearly shows the Intel CPU performing better than the Vectorblox MXP overlay. On the contrary, when considered OPC, the Vectorblox MXP overlay defeats all other platforms by a considerably large margin as shown in the graph. The high OPC of the Vectorblox MXP overlay can be attributed to its efficient implementation on the FPGA fabric and use of HP ports on the Zynq for communication.

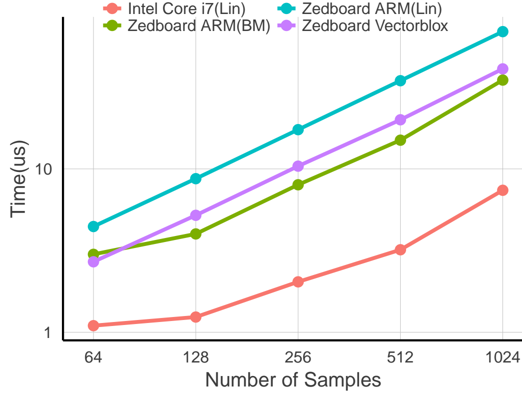


Figure 4.3: C Execution Time for 1D-FIR

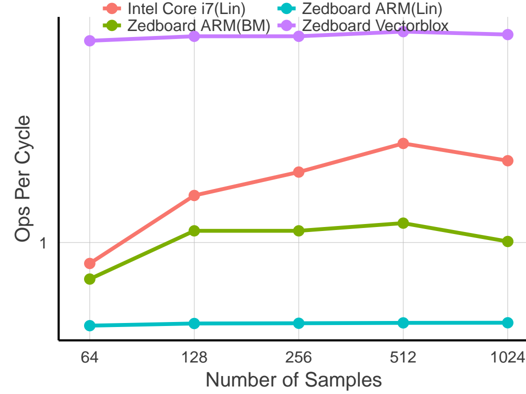


Figure 4.4: C OPC for 1D-FIR

Table 4.4: Execution time of 12-Tap FIR Filter (C) on different platforms

Samples	**Intel Core i7**	**ARM BM**	**ARM VBX**	**ARM Linux**
64	1.097	3	2.7	4.439
128	1.239	4	5.2	8.718
256	2.035	8	10.4	17.4
512	3.2	15	20	34.69
1024	7.4	35	41	69.305

Table 4.5: 12-Tap FIR Filter(C Implementation) OPC on different platforms

Samples	**Intel Core i7**	**ARM BM**	**ARM VBX**	**ARM Linux**
64	0.838	0.735	5.451	0.497
128	1.485	1.103	5.661	0.506
256	1.808	1.103	5.661	0.507
512	2.3	1.177	5.888	0.508
1024	1.989	1.008	5.744	0.509

4.2 2D Convolution

2D Convolution is a matrix operation that involves processing a large number of data samples usually pixels of an image and is an application that needs to be done in real time. The results obtained when the 2D Convolution application was accelerated using OpenCL for different image sizes is shown in figures 4.5 and 4.6.

OpenCL Implementation:

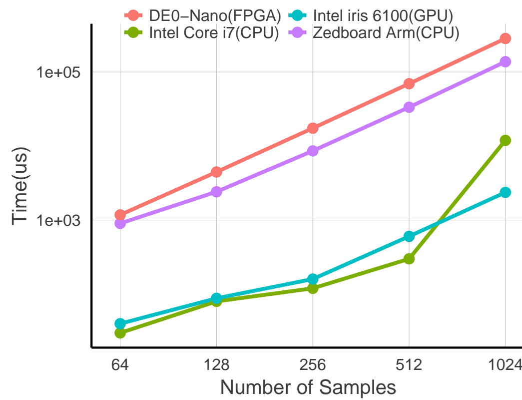


Figure 4.5: OpenCL Execution Time for 2D Convolution

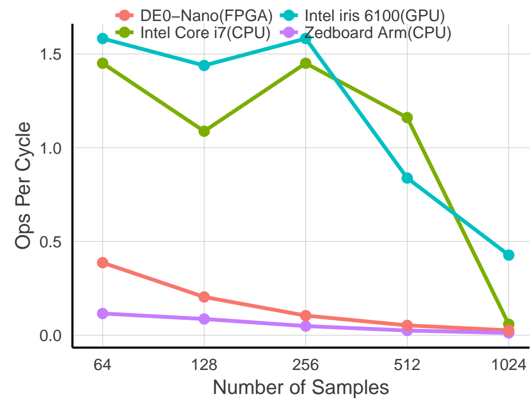


Figure 4.6: OpenCL OPC for 2D Convolution

Table 4.6: 2D Convolution(OpenCL) Execution Time on different Platforms

Samples	**Intel Core i7**	**Intel Iris 6100**	**Zedboard ARM v7**	**DE0-NANO FPGA**
64	30	40	903	1178
128	80	88	2415	4470
256	120	160	8607	17442
512	300	604	3346	69561
1024	11944	2370	137799	238270

In the timing performance as shown in figure 4.5 the intel core i7 CPU and GPU perform almost the same, this could be attributed to the increased number of computations involved in convolution. The FPGA performs the worst in the case of timing

Table 4.7: 2D Convolution(OpenCL) OPC on different platforms

Samples	Intel Core i7	Intel Iris 6100	Zedboard ARM v7	DE0-NANO FPGA
64	1.450	1.582	0.1156	0.3867
128	1.088	1.4386	0.0864	0.2038
256	1.4506	1.5825	0.0485	0.1044
512	1.1605	0.838	0.0249	0.0523
1024	0.0582	0.4273	0.0121	0.0257

performance due to the high amount of communication time to transfer data in and out of the FPGA. If we look at Ops per cycle in figure 4.6, the GPU, and CPU perform really well for image sizes lower than 256x256 but with higher image sizes all platforms seem to be processing fewer operations per cycle. This could be attributed to the higher time taken for data movement when larger sample sizes are processed.

C implementation:

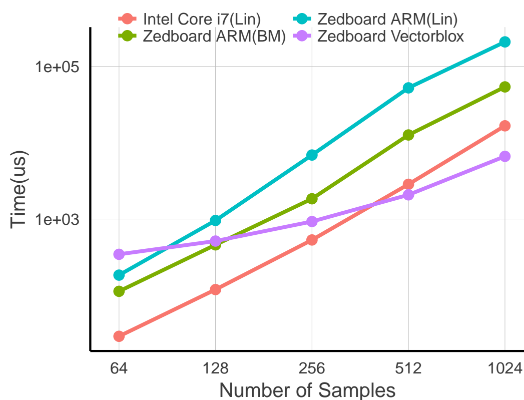


Figure 4.7: C Execution Time for 2D Convolution

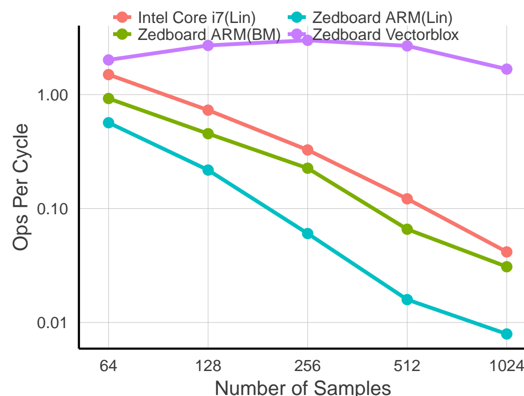


Figure 4.8: C OPC for 2D Convolution

The results of the 2D convolution benchmarks are really interesting primarily due to the performance demonstrated by the vectorblox matrix processor. If we see figure 4.7 every other hardware platform scales up in computation but the vectorblox processor takes more time than the other platforms when the sample count is low but

Table 4.8: 2D Convolution(C Implementation) Execution Time on Various Platforms

Samples	Intel Core i7	ARM BM	ARM VBX	ARM Linux
64	29	112.82	344.96	184
128	119	459.1	514.68	958
256	532	1845.612	928.134	6912
512	2858	12673.48	2077.785	52555
1024	16706	54087.19	6643.45	210588

Table 4.9: 2D Convolution(C Implementation) OPC on Various Platforms

Samples	Intel Core i7	ARM BM	ARM VBX	ARM Linux
64	1.500	0.925	2.018	0.567
128	0.731	0.454	2.705	0.217
256	0.327	0.226	3.009	0.0604
512	0.121	0.065	2.681	0.015
1024	0.041	0.030	1.676	0.007

as the sample count is increased the vectorblox starts performing much better than the other platforms. In Ops per cycle performance shown in figure 4.8 as well the vectorblox seems to be churning out far more Ops per cycle as compared to the other platforms. This behavior could again be attributed to the highly efficient design of the Vectorblox engine.

4.3 Kmeans

The Kmeans algorithm is a unique algorithm as its a map reduce type computation. The specialty of a map-reduces algorithm is that it has a lot of branching which tends to break the parallelism when accelerated using parallel threads in OpenCL.

OpenCL Implementation:

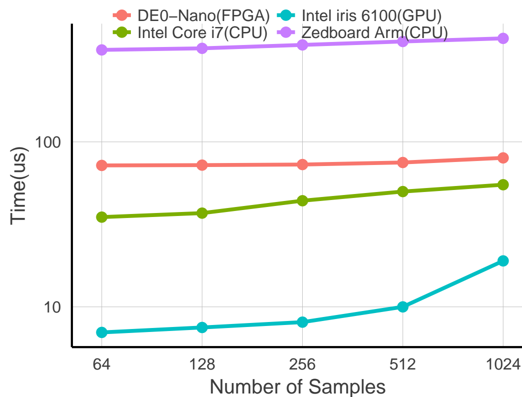


Figure 4.9: OpenCL Execution Time for Kmeans

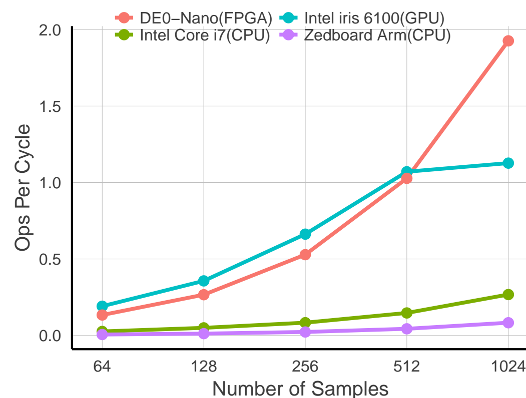


Figure 4.10: OpenCL OPC for Kmeans

Table 4.10: Kmeans(OpenCL) Execution Time on Various Platforms

Samples	**Intel Core i7**	**Intel Iris 6100**	**Zedboard ARM v7**	**DE0-NANO FPGA**
64	35	7	361	72
128	37	7.5	369	72.3
256	44	8.08	387	72.9
512	50	10	406	75
1024	55	19	424	80

As you can see when executed with OpenCL the GPU still performs the best as it has multiple threads spawned but the CPU and FPGA performance is really low, this could be attributed to the single threaded operations on the CPU and the low clock frequency of the FPGA. On comparing operations per cycle, it's clearly seen that the

Table 4.11: Kmeans(OpenCL) OPC on Various Platforms

Samples	Intel Core i7	Intel Iris 6100	Zedboard ARM v7	DE0-NANO FPGA
64	0.026	0.1911	0.00611	0.1337
128	0.0497	0.3568	0.0119	0.2664
256	0.0836	0.6624	0.02281	0.5284
512	0.1472	1.0705	0.0434	1.0273
1024	0.267	1.126	0.083	1.926

CPU performance does not scale with increasing number of operations but in the case of the GPU and FPGA scale up in operations per cycle and the DE0-NANO FPGA scales the highest with increasing samples to be processed. The FPGA performance in the case of operations per cycle can be attributed to highly efficient kernel design routed to the fabric.

C implementation:

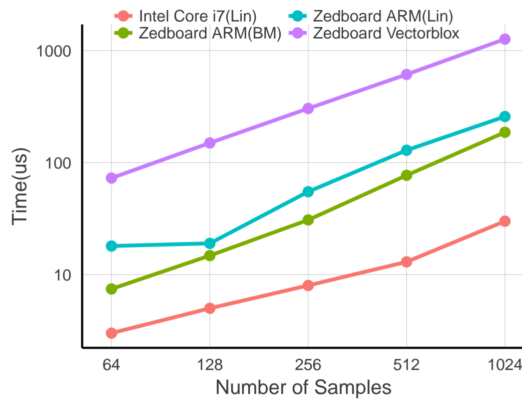


Figure 4.11: C Execution Time for Kmeans

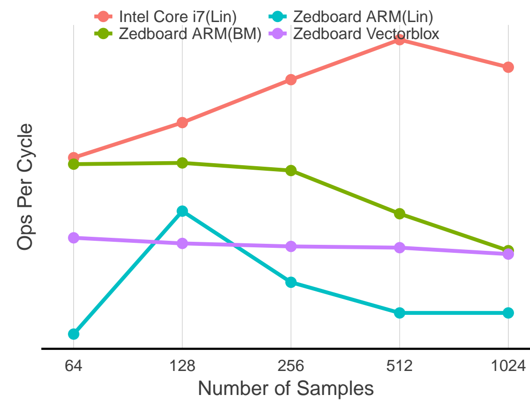


Figure 4.12: C OPC for Kmeans

As mentioned earlier since Kmeans is a map reduce algorithm, it tends to break parallelism after a few stages and hence as seen in figure 4.11 the intel CPU performs the best in this case due to lack of parallelism in the application. A similar trend is seen when Ops per cycle of the C implementation is compared.

Table 4.12: Kmeans (C Implementation) Execution Time on Various Platforms

Samples	**Intel Core i7**	**ARM BM**	**ARM VBX**	**ARM Linux**
64	3	7.44	72.79	18
128	5	14.79	149.898	19
256	8	30.774	304.512	55
512	13	77.037	612.924	129
1024	30	186.666	1267.569	258

Table 4.13: Kmeans (C Implementation) OPC on Various Platforms

Samples	**Intel Core i7**	**ARM BM**	**ARM VBX**	**ARM Linux**
64	0.306	0.296	0.202	0.122
128	0.368	0.298	0.196	0.232
256	0.46	0.286	0.193	0.160
512	0.566	0.229	0.192	0.136
1024	0.490	0.189	0.185	0.136

4.4 ATAX

The ATAK kernel is a linear algebra kernel which is a part of the Poly Bench benchmark suite and is written in OpenCL and calculates the following equation:-

$$y = A^T AX \tag{4.1}$$

As seen in equation 4.1 the computation involves a lot of operations involving matrix transpose and matrix multiplications and hence can be categorized as compute intensive in nature. Thus accelerating such an algorithm makes a lot of sense. The application was run on the previously mentioned hardware platforms and the results are shown in figure 4.13 and 4.14.

OpenCL Implementation:

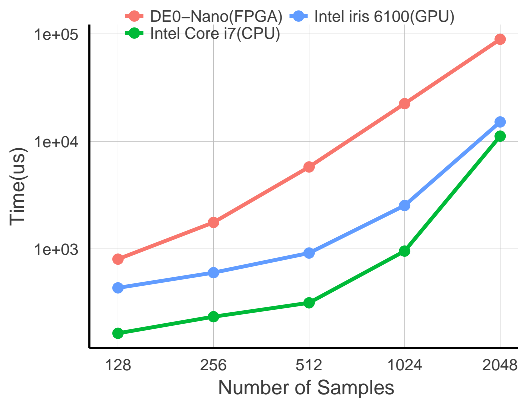


Figure 4.13: OpenCL Execution Time for ATAX

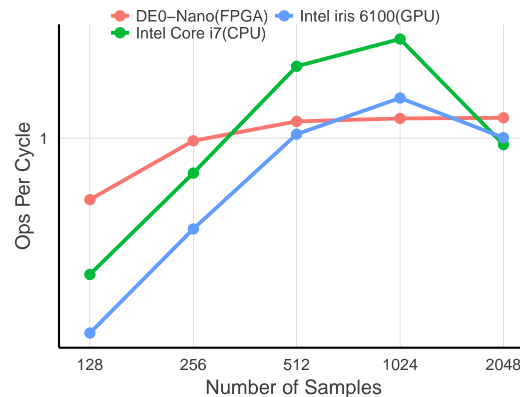


Figure 4.14: OpenCL OPC for ATAX

As we can see from the timing performance shown in figure 4.13 the CPU performs better than both the GPU and FPGA, this could be attributed to the high clock frequency at which the CPU is operating. The FPGA is really slow here and its stunted performance can be attributed to the much lower frequency at which it is operating. If we look at the operations per cycle performance both the CPU and GPU follow a kind of a bell curve pattern where they begin by computing much lesser operations per cycle for lower samples, reach a peak of operations per cycle

Table 4.14: ATAX(OpenCL) Execution Time on Various Platforms

Samples	Intel Core i7	Intel Iris 6100	DE0-NANO FPGA
128	164	434	803
256	234	601	1762
512	315	915	5782
1024	954	2533	22453
2048	11197	15172	89182

Table 4.15: ATAX(OpenCL) OPC on Various Platforms

Samples	Intel Core i7	Intel Iris 6100	DE0-NANO FPGA
128	0.249	0.137	0.534
256	0.7001	0.396527	0.973476349
512	2.080507937	1.041804272	1.186624232
1024	2.747840671	1.505331084	1.222297476
2048	0.936479414	1.00527383	1.230932037

following which they drop down again. The FPGA, on the other hand, begins with a much higher operations per cycle output and remains constant with increase in samples and scales in a slightly linear fashion and in the end still ends up producing more operations per cycle as compared to the CPU and GPU, again this can be attributed to the highly specialized hardware implementation of the kernel on the FPGA.

C implementation:

The C Implementation of the atax algorithm was done on hardware platforms and the results are as shown in figure 4.15 and 4.16. In the timing performance the Intel CPU performs the best while the ARM CPU on the DE)-NANO performs better than the ARM CPU on the zedboard, this can be attributed to the higher clock frequency of the arm processor on the DE0-NANO board. The intel CPU also outperforms the others in terms of Ops per cycle as shown in figure 4.16.

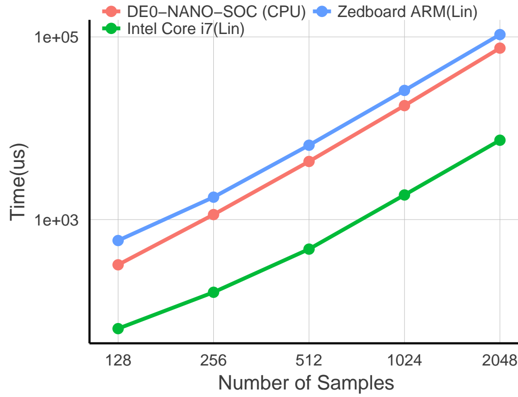


Figure 4.15: C Execution Time for ATAX

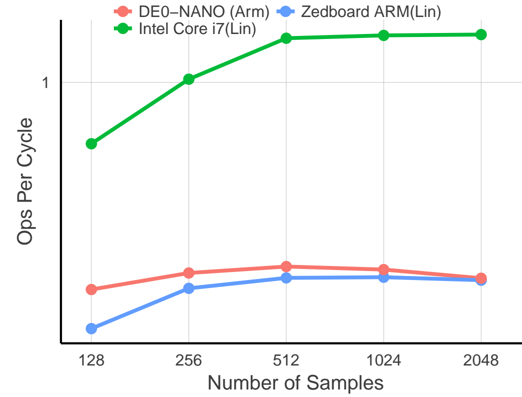


Figure 4.16: C OPC for ATAX

Table 4.16: ATAX (C Implementation) Execution Time on Various Platforms

Samples	**Intel Core i7**	**Zedboard ARM**	**DE0-NANO ARM**
128	64	590	320
256	160	1760	1135
512	475	6525	4332
1024	1861	25970	17714
2048	7400	106151	75466

Table 4.17: ATAX (C Implementation) OPC on Various Platforms

Samples	**Intel Core i7**	**Zedboard ARM**	**DE0-NANO ARM**
128	0.64	0.166	0.221
256	1.024	0.2233	0.249
512	1.379	0.240	0.261
1024	1.408	0.242	0.255
2048	1.416	0.236	0.240

4.5 BICG

The biconjugate gradient method is an algorithm to solve systems of linear equations of types as shown :-

$$Ax = b \tag{4.2}$$

Unlike the conjugate gradient method, this algorithm does not require the matrix "A" to be self-adjoint, but instead one needs to perform multiplications by the conjugate transpose.

The BICG algorithm was implemented in OpenCL and C and the results are as shown below.

OpenCL Implementation:

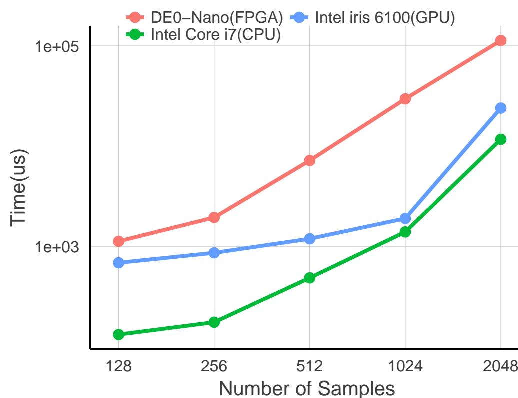


Figure 4.17: OpenCL Execution Time for BICG

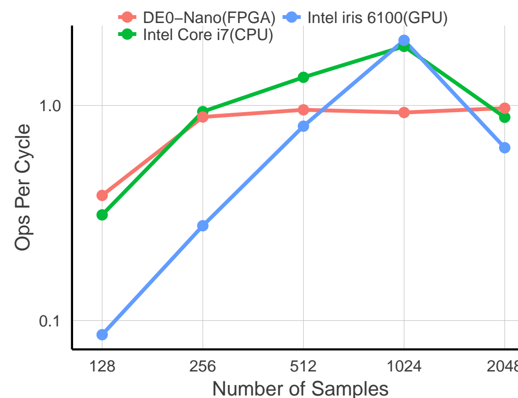


Figure 4.18: OpenCL OPC for BICG

The performance, as well as operations per cycle results for BICG, are pretty similar to the ATAX performance results, this can be attributed to the similarity in the type of operations performed being matrix multiplications. In the case of BICG as well the CPU and GPU perform much better in the case of timing operations while the FPGA take much more time for computation which can be attributed to its low frequency of operation. In Operations per cycle performance, the CPU and GPU again take a bell curve appearance with the lesser number of operations per cycle produced when sampling size increases while the FPGA Operations per cycle scales

Table 4.18: BICG(OpenCL) Execution Time on Various Platforms

Samples	Intel Core i7	Intel Iris 6100	DE0-NANO FPGA
128	132	686	1123
256	175	862	1940
512	485	1190	7187
1024	1393	1899	29595
2048	11685	23950	112997

Table 4.19: BICG(OpenCL) OPC on Various Platforms

Samples	Intel Core i7	Intel Iris 6100	DE0-NANO FPGA
128	0.31030303	0.086848662	0.381848915
256	0.936228571	0.276464881	0.884157385
512	1.351257732	0.801051184	0.954648853
1024	1.881866475	2.007900809	0.92732709
2048	0.897369277	0.63682733	0.9715035

almost constantly with an increase in samples.

C implementation:

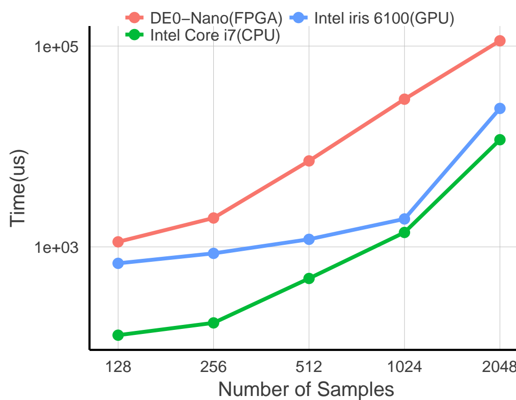


Figure 4.19: C Execution Time for BICG

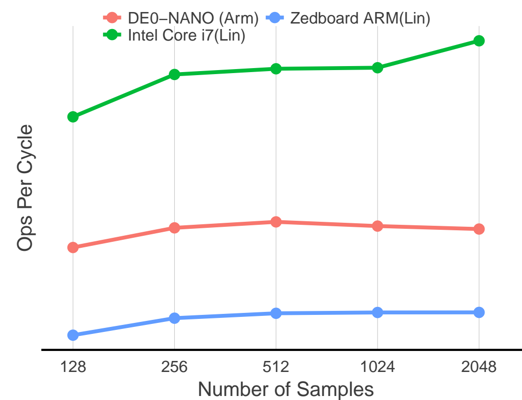


Figure 4.20: C OPC for BICG

As mentioned in the previous section due to the application similarity to the

Table 4.20: BICG (C Implementation) Execution Time on Various Platforms

Samples	**Intel Core i7**	**Zedboard ARM**	**DE0-NANO ARM**
128	88	627	292
256	285	2304	1059
512	1108	8995	4112
1024	4409	35833	16795
2048	15419	143273	68177

Table 4.21: BICG (C Implementation) OPC on Various Platforms

Samples	**Intel Core i7**	**Zedboard ARM**	**DE0-NANO ARM**
128	0.465	0.156	0.242
256	0.574	0.170	0.267
512	0.591	0.174	0.275
1024	0.594	0.175	0.269
2048	0.680	0.175	0.266

ATAX application the performance benchmarking look almost similar as shown in figures 4.19 and 4.20. The Intel CPU performs better than the other platforms in both cases.

4.6 Summary

To summarize the results of these experiments, we have the key takeaways from each of the benchmark applications as below.

12-Tap FIR Filter: The iris GPU gives much lower execution time as compared to the FPGA for the FIR filter operation but the FPGA overtakes all the other devices with higher OPC with 2x greater OPC as compared to the iris GPU. This could be attributed to the fact that the FPGA runs a fully pipelined customized datapath which does not suffer from the overheads of software scheduling that takes place in case of the GPU. In the case of C Implementation, the Vectorblox gives the highest OPC which is almost 5x greater than the ARM processor.

2D Convolution: The Vectorblox MXP processor overtakes all other platforms in both execution time as well as OPC due to its highly efficient vector lane architecture. This benchmark clearly shows that image processing tasks which require a lot of SIMD operations, the MXP is the best bet. The AOCL implementation on the DE0 performs much slower here due to the nature of the application which involves a number of data transfers to and from the FPGA.

Kmeans: Since Kmeans is an application where SIMD parallelism is broken due to the nature of the application. This is the primary reason why the Vectorblox MXP and the GPU do not perform that well. The CPU performs the best here. An interesting fact in Kmeans is that the execution time on the DE0-NANO FPGA is 15x faster than the vectorblox MXP, this clearly shows that the MXP processor is really not good when the computation cannot be expressed in a SIMD form.

ATAX: The ATAX application which is more sequential in nature performs extremely well in the CPU. Even though a dedicated datapath is created on the FPGA, it still fails to outperform the CPU, this could be attributed to the low operating frequency of the DE0-NANO.

BICG: The performance, as well as OPC results for BICG, are pretty similar to the ATAX performance results, this can be attributed to the similarity in the type of operations performed being matrix multiplications. In the case of BICG as well the CPU and GPU perform much better in the case of timing operations while the FPGA take much more time for computation which can be attributed to its low frequency of operation.

As seen in the application benchmarks in this chapter, it's evident that hardware acceleration is certainly possible by means of software like abstractions like C and OpenCL. Efforts made by Vectorblox and AOCL in the domain promotes this notion of hardware design using software abstractions. Advantages of these abstractions are that they enable software engineers to make hardware designs in a simple and easy manner. Unlike methods using HDL which take a steep learning curve to master, these abstractions have a relatively smaller learning curve and hence improve design productivity to a great extent. It is also seen from the experiments that not all applications result in an accelerated version when ported to a hardware design, this could be one of the pitfalls of an abstracted hardware design also the software abstractions result in some degree of reduced performance as compared to designing HDL-based hardware with a fully customized data path, for example, software abstractions severely limit the maximum frequency with which we can run the hardware. Acceleration with fixed hardware devices like GPU result in a great performance for floating point operations but its to be noted that we might not always need floating point accuracy at all times this results in GPU hardware being left idle. Nevertheless, this approach could be great especially when the application is of such a nature where the high-level design is constantly changing. In these cases maintaining HDL and constant modifications might be a challenging task and hence having high-level software abstractions make sense.

Chapter 5

Hardware Virtualization And Lab On The Cloud

As we saw in the previous chapters, there are a number of ways in which hardware acceleration can be made possible, and these methods are only a few that have been explored till now. There is a great need for design space exploration in this area and only with extensive experimentation and demand for these tools and methods can the efficiency of these methods be improved. One approach is to provide easy and inexpensive access to these hardware platforms in an abstracted manner to their target customers to increase the popularity of these methods of hardware acceleration. Hardware virtualization is the way to go where computing platforms should be abstracted from the user and can be accessed remotely for the acceleration of compute-intensive applications. Giving end customers a real feel of coding on the actual hardware is a much more convincing method of getting them to adopt a technology than giving them a presentation or demo about it.

5.1 Existing Work

There have been a number of cases where hardware virtualization has already been used especially in the field of high performance computing as mentioned in section 5.2.

One of the service providers of FPGA based virtualized hardware on cloud is SciEngines [29]. SciEngines offers a number of reconfigurable computing (RC) hardware for large-scale scientific analysis. Modules with 1 to 8 FPGAs each are combined to form computers and clusters with a number of FPGAs for high performance computing (HPC) applications [29]. Having an FPGA on a server being used for HPC is great but this demands the involvement of a PC to continually reconfigure the whole FPGA with a new bitstream. But now, with the application of the Xilinx partial-reconfiguration technology, its feasible to design FPGA-based clients for a distributed computing network. A team at the Hamburg University of Applied Sciences created a prototype for such a client and implemented it in a single FPGA [30]. Also with the recent acquisition of major FPGA manufacturer Altera by Intel, there have been plans for the intel Xeon server chip to ship with an inbuilt FPGA. Intel sees FPGAs as the key to designing a new generation of products to address emerging customer workloads in the data center sector, as well as the Internet of Things (IoT) [31]. Though all these existing developments are quite interesting most of them focus on virtualizing FPGA hadware for HPC. Most of the hardware is high power and bulky units of hardware. There hasnt been much development in virtualizing hardware for the embedded market. Section 5.2 tries to make the case for virtualizing embedded hardware devices to the cloud and its key areas of application.

5.2 The Case for Embedded Hardware Virtualization

Virtualization hides the physical characteristics of a computing platform from the users, presenting instead another abstract computing platform[32]. Hardware platforms are generally expensive and to explore and assess a set of platforms for a particular application might end up becoming an expensive task. Thus in this chapter, we have explored virtualization of embedded hardware platforms.

One of the applications for hardware virtualization is in the field of academic labs. In most labs, large chunks of lab room, real estate, staff and dedicated resources are

allocated to maintain hardware labs in colleges, universities. This could result in really high installation and maintenance costs for these labs. Virtualizing these labs or putting these "Labs on the Cloud" would result in a great reduction of costs. There could be central servers to which a farm or cluster of hardware platforms could be connected to a cloud server and people could log into them from a remote machine and use the hardware attached to the server. One existing example of this type of virtualization is dispatch of heavy compute jobs to high-performance servers which can be rented at a small fee such as Google cloud engine, Amazon AWS. Extending this concept to embedded computing devices would change the paradigm of hardware labs and also could be applied to product selection and deployment analysis by commercial vendors as well. The basic idea of virtualizing embedded hardware platforms by making use of the cloud is highlighted in figure 5.1.

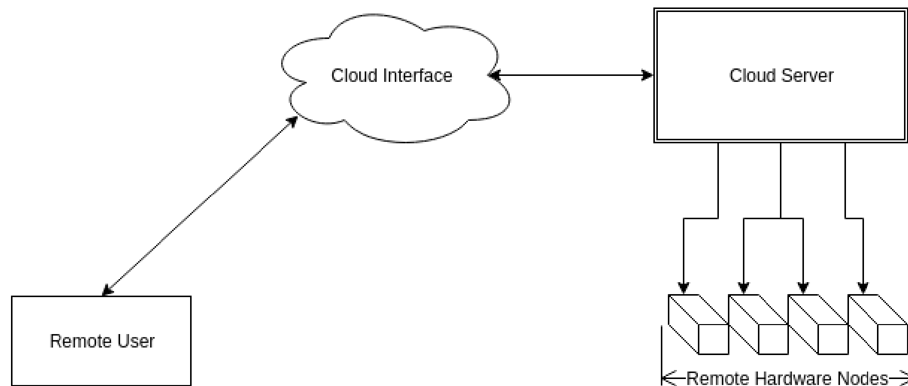


Figure 5.1: Basic Hardware Virtualization Using Cloud

5.3 The Cloud9 based virtualization for Embedded Hardware

As seen in the experiments sections a lot of benchmarks have been performed on the Vectorblox MXP overlay on the Zedboard. To assess the ease of setup and use of hardware virtualization, we choose the Zedboard as the target device that would be virtualized. In order to put the Zedboard online, we connected it to an Ubuntu Server

which acted as the host for the Zedboard connected which acted as the slave. The server-device setup is shown in figure 5.2.

Figure 5.2: The Zedboard-Server Setup

5.3.1 The Cloud9 IDE

Cloud9 IDE is an online integrated development tool, published as open source software from its version 3.0. It supports a lot of programming languages, including PHP, Ruby, Perl, Python with Node.js, and Go. In our case, we use only C/C++. It enables developers to get started with their coding immediately with pre-configured cloud-based workspaces, and the web development features like a live preview of the code and web browser compatibility and testing [33]. The Cloud9 IDE source code was used to spawn a server based on node.js [33] and the node server is used to graphically display all the project workspaces and code that is required to program the Vectorblox MXP overlay instantiated on the FPGA at the Zedboard.

5.3.2 Node.js and the Cloud9 Server Setup

The Cloud9 IDE is entirely built in javascript and can be hosted online by spawning a node server. The node server that is spawned makes the Cloud9 IDE available online in the form of a web application. In order to spawn the node server, node.js needs to be installed in the system where the node server is intended to be spawned. Node.js can be installed from its official website : https://nodejs.org

In order to spawn a server in node.js we need to prepare a server script and bind it to the web application that we need to put online, in case of Cloud9 the server script can be found along with their web application source code in :- https://github.com/c9/core. The following code in figure 5.3 is a bare minimum structure of a node.js server code that makes use of HTTP ports.

```
var http = require('http');
var fs = require('fs');
var url = require('url');


// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // HTTP Status: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         //Page found
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

         // Write the content of the file to response body
         response.write(data.toString());
      }
      // Send the response body
      response.end();
   });
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Figure 5.3: Basic Node Server Script

As seen in the code snippet, the node server listens on a HTTP port which is generally accessed by an IP address and when any request comes on that IP address it re-directs the request to the Cloud9 Web application in the background.

The node server is spawned on the host Ubuntu machine via terminal as shown in figure 5.4.

```
prashant014@Ubuntu23:~/c9sdk$ node server.js -l 172.21.145.146 -w ~/mxp/mxp_c9/examples/software/bmark/ --username username --password password
Starting standalone
Authentication is required when not running on localhost.
If you would like to expose this service to other hosts or the Internet
at large, please specify -a user:pass to set a username and password
(or use -a : to force no login).
Use --listen localhost to only listen on the localhost interface and
and suppress this message.

Connect server listening at http://127.0.0.1:8181
CDN: version standalone initialized /home/prashant014/c9sdk/build
Started '/home/prashant014/c9sdk/configs/standalone' with config 'standalone'!
Cloud9 is up and running
```

Figure 5.4: Spawning The Node Server

5.3.3 The Lab On Cloud Web Application

Once the node server is started, and the IP address of the node server is online, a web page is generated by the Cloud9 IDE which can be viewed by going to the IP address at which the node server has been spawned. The screen-shot of the web page generated has been shown in figure 5.5.

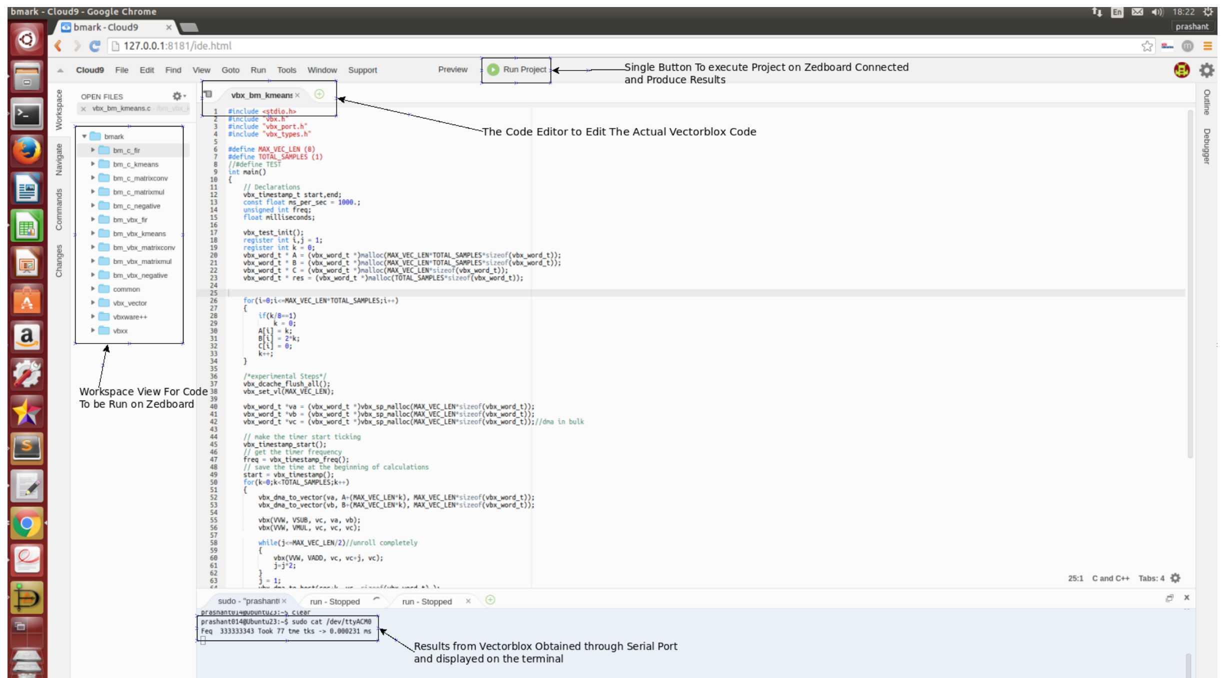


Figure 5.5: The Cloud9 IDE Webpage

As seen in the screen-shot in figure 5.5 the Cloud9 IDE makes it really easy to program and write code for hardware in remote servers via a web browser. As you can see in the web browser, there is a window present to edit code and a sidebar with all the project folders where different projects are present that can be run on the MXP overlay. Once the code is finalized, we can use the run button as shown in figure 5.5 to run the code. This action triggers an internal script in the node server which then triggers the compilation of the code on the Ubuntu server followed by the transfer of the executable binary to the Zedboard. Once the Zedboard finishes the computation, the results are read back through the serial port and displayed in the results window as shown in figure 5.5.

Apart from setting up workspaces, the online web application can also be used to customize various project settings as shown in figure 5.6.

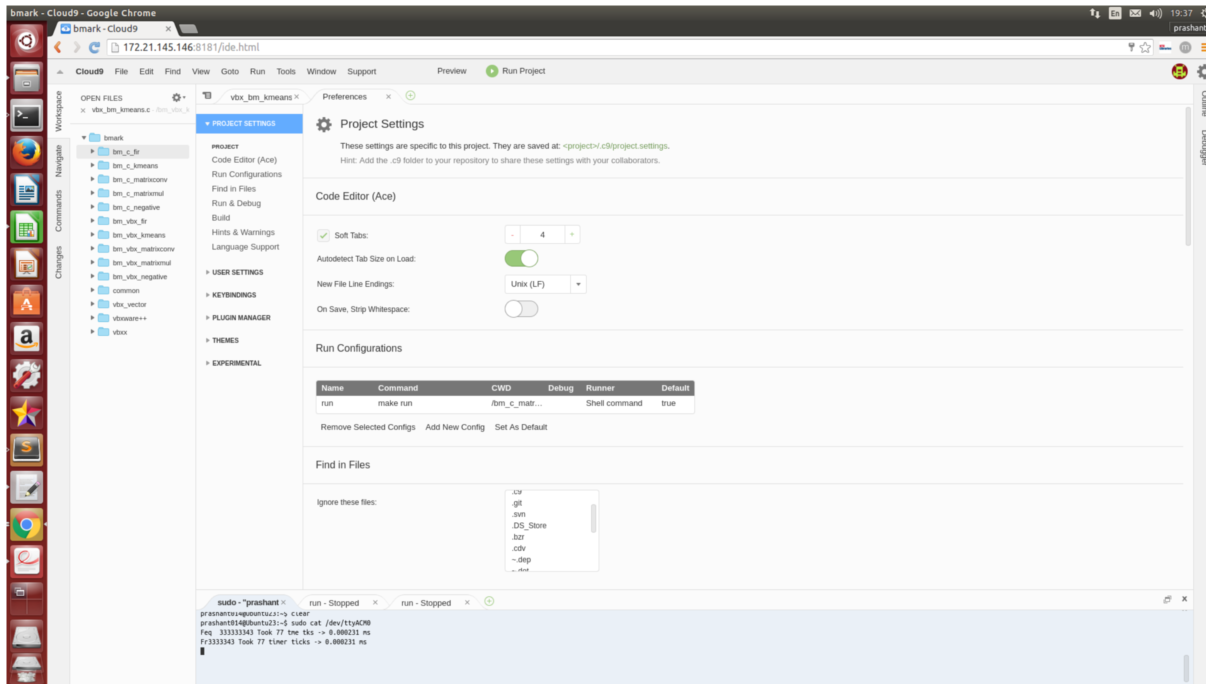


Figure 5.6: The Cloud9 IDE project customization

Custom project settings gives the advanced users freedom to design their own build system for the project that they run and also put in compiler optimizations etc.

Thus this method of hardware virtualization using web pages could give access to hardware from anywhere and could be used by researchers and academia for course labs as well as research experiments. The applications could also be extended to commercial vendors who would like to test their applications on multiple embedded hardware targets.

Chapter 6

Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

6.1 Conclusions

This report proposed an approach for performing hardware acceleration by means of software based high-level design (using MXP overlay and Altera OpenCL framework) and quantifying performance benefits. The first case study in chapter 3.1 shows us that by merely using compiler optimizations it is possible to accelerate the applications on an embedded device, Xilinx Zynq since compiler optimization enable the use of SIMD NEON engine. From the data tabulated in chapter 3.1 we observe $\approx 20\times$ improvement in the performance of the ARM processor by merely turning on the compiler optimizations (enabling the use of SIMD NEON engine). We observe a performance of up to 550 MOPS for NEON SIMD engine which is calculated based on the fact that 1K samples are processed using a 12-tap filter (12 multiplication and 11 addition operations per sample) in 53 μs, in other words, 23K operations are performed in 53 μs. Although DSP overlay (given a high-speed communication interface around the overlay) can provide a performance of 700 MOPS, 1500 MOPS and 2300 MOPS in the case of 4, 8 and 12-tap filter, respectively, the performance of the design in [26] is limited by the use of low performance general purpose (GP)

port for data communication. We believe that a high-speed communication interface around DSP overlay would be able to give us 4× performance speed-up compared to SIMD NEON engine. We also observed that the MXP overlay performs 20% better than the SIMD NEON engine. The small gain in the performance is due to the use of a small number of data samples. MXP overlay generally performs well when the number of data samples to be processed is high. The case-study demonstrated that FPGA-based overlays (MXP and DSP overlay) have the potential to outperform hard vector processing engine (SIMD NEON). However, the performance benefits would be satisfactory when processing a large amount of data samples including a high-speed communication interface.

The second case study in chapter 3.1 demonstrated dynamic reconfiguration of the FPGA by use of the Altera OpenCL runtime. The dynamic reconfiguration case could be very useful when there are multiple hardware designs that need to be accelerated on the FPGA but there is not enough area in the fabric to fit them all. We conclude that the vectorblox MXP overlay is much better at handling this type of workloads with ease as there is no need of actual reconfiguration of the FPGA.

This work included developing an understanding of compute kernels and their benchmarking using hardware acceleration on overlay and GPU architectures with detailed performance evaluation of these devices including AOCL generated accelerators and commercial multi-core devices. Use of programming abstractions for hardware design have helped improve the design productivity and example designs were experimented with for a set of kernels. Experiments were designed with a number of compute kernels and were accelerated using Overlay architectures and AOCL generated accelerators. A performance comparison was done for these algorithms when run on different platforms and it was analyzed how hardware acceleration affects the performance in each case.

Finally, this thesis concludes with an exotic demonstration of hardware virtualization by using the cloud. With all these methods of hardware acceleration, virtualization of hardware could be a great method to give access to a large community of developers to experiment with and learn the alternative methods of hardware acceleration.

6.2 Future work

Some of the main future research directions that could be considered are towards making this technology easier and more portable across devices. Some of the key areas have been listed below:

- **Mature firmware and drivers to integrate overlays with the host processor**: In order to use the various overlay architectures efficiently, it is necessary to interface them properly with a host processor system, drivers, and firmware that control the overlay must work really fast in order to avoid performance drops due to communication.
- **OpenCL Support for overlays**: Dynamic loading of OpenCL kernels to overlays by providing OpenCL support for overlays.

Finally, with these initiatives like hardware virtualization, it is possible to reach hundreds of thousands of developers worldwide and give them a platform and framework to program and code such devices and also develop their skills which would eventually lead to a paradigm shift of many people adopting these methods.

Bibliography

[1] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of todays high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[2] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on DSP blocks," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015.

[3] Software acceleration in the zynq-7000 epp. [Online]. Available: www.avnet.com

[4] C. V. Dobson, "An architecture study on a xilinx zynq cluster with software defined radio applications," Ph.D. dissertation, Virginia Tech, 2014.

[5] A. George, H. Lam, and G. Stitt, "Novo-g: At the forefront of scalable reconfigurable supercomputing," *Computing in Science Engineering*, vol. 13, no. 1, pp. 82–86, 2011.

[6] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, pp. 60–65, 2011.

[7] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner, "Theoretical and practical limits of dynamic voltage scaling," in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 868–873.

[8] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[9] O. Albaharna, P. Y. K. Cheung, and T. Clarke, "On the viability of FPGA-based integrated coprocessors," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1996, pp. 206–215.

[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[11] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581*, 2010.

[12] Wikipedia. (2016, feb) Opencl. [Online]. Available: https://en.wikipedia.org/wiki/OpenCL

[13] K. O. W. Group *et al.*, "The opencl specification," *Version*, vol. 1, no. 29, p. 8, 2008.

[14] D. D. Gajski and R. H. Kuhn, "Guest editors' introduction: New vlsi tools," *Computer*, vol. 16, no. 12, pp. 11–14, 1983.

[15] L. Wirbel, "Xilinx sdaccel: a unified development environment for tomorrows data center," Technical Report, The Linley Group Inc, Tech. Rep., 2014.

[16] S. Altera, "Altera sdk for opencl," *Altera, Corp*, 2014.

[17] A. Brant and G. Lemieux, "ZUMA: an open FPGA overlay architecture," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 93–96.

[18] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture," in *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011.

[19] C. Plessl and M. Platzner, "Zippy - a coarse-grained reconfigurable array with support for hardware virtualization," in *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2005, pp. 213–218.

[20] N. W. Bergmann, S. K. Shukla, and J. Becker, "QUKU: a dual-layer reconfigurable architecture," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, pp. 63:1–63:26, Mar. 2013.

[21] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2010, pp. 13–22.

[22] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[23] C. Liu, C. Yu, and H. So, "A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, pp. 228–228.

[24] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[25] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[26] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell, "Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform," *Journal of Signal Processing Systems*, vol. 77, no. 1–2, pp. 61–76, Oct. 2014.

[27] A. Severance and G. G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[28] L. Pouchet, "Polybench: The polyhedral benchmark suite (2011), version 3.2," 2011.

[29] SciEngines. (2015, feb) Reconfigurable high-performance computing solutions. [Online]. Available: http://sciengines.com

[30] X. J. I. 79. (2015, jun) Hamburg university: Cloud computing. [Online]. Available: http://www.xilinx.com/about/customer-innovation/data-centers-storage-hpc/cloud-computing.html

[31] R. Miller. (2015, jun) Intel begins rollout of fpgas to power the cloud, iot. [Online]. Available: http://datacenterfrontier.com/intel-begins-rollout-of-fpgas-to-power-the-cloud-iot/

[32] I. G. Education, "Virtualization in education," *IBM Corporation, Whitepaper*, 2007.

[33] C. 9. (2016, feb) Cloud 9 ide. [Online]. Available: https://github.com/c9/core