Porting and Evaluation of Overlay Architectures for FPGAs with Scientific Kernels

Konstantinos Gkougkoulias

CE-MS-2017-18

Abstract

In recent years due to the slow down of Moores Law and Dennard Scaling, alternative architectures are starting to be used instead of plain CPU implementations. These new architectures, such as FPGAs and GPUs, offer higher performance to power consumption ratio when compared with a CPU only implementation. But these new approaches have to sacrifice programmability in favor of performance gains. While GPUs are somewhat easily programmable and provide high performance this comes at the cost of high power consumption. FPGA programming on the other hand is a tedious and time consuming task. Specialized personnel is required for this, as their programming requires a background in designing with HDL languages. Furthermore an implementation is specific to a certain algorithm and cannot be used for any other algorithm even if it is slightly different. So if a new algorithm for a particular task is found then a part of the design process has to be redone. Also designing for FPGAs is a computationally intensive task as the whole design after simulation has to be synthesized and then placed and routed (P&R) for a particular FPGA every time the design changes slightly. This process of mapping the design can take hours or even days to compute for large designs. In recent years developments in High Level Synthesis (HLS) and OpenCL have made the whole process of designing for FPGAs an easier task. But this solution is not without problems either as the algorithm has to still be implemented for a specific FPGA device. A solution to the FPGA synthesis and P&R problem has recently been proposed with the name of FPGA Overlay Architectures. The core concept of this idea to abstract the FPGA create a virtual FPGA on top of the underlaying physical one in order to help with configuration and compile time. In this thesis, we investigate available alternative overlay architectures and select the most appropriate architecture for our analysis. We extended the selected architecture to be deployed on alternative FPGA hardware and to work in a shared CPU/FPGA system. Then, we implemented a number benchmarks to evaluate various aspects of system performance. Our results show that our architecture can be reconfigured in only 11.9us, as compared to seconds for full FPGA reconfiguration. However, the overlay architecture uses 10.5x more LUTs and causes a drop in frequency of about 30% for the chosen architecture. For future work, there is room to improve these results by optimizing the interconnect network of the device.

Porting and Evaluation of Overlay Architectures for FPGAs with Scientific Kernels

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Konstantinos Gkougkoulias
born in Larisa, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Porting and Evaluation of Overlay Architectures for FPGAs with Scientific Kernels

by Konstantinos Gkougkoulias

Abstract

In recent years due to the slow down of Moores Law and Dennard Scaling, alternative architectures are starting to be used instead of plain CPU implementations. These new architectures, such as FPGAs and GPUs, offer higher performance to power consumption ratio when compared with a CPU only implementation. But these new approaches have to sacrifice programmability in favor of performance gains. While GPUs are somewhat easily programmable and provide high performance this comes at the cost of high power consumption. FPGA programming on the other hand is a tedious and time consuming task. Specialized personnel is required for this, as their programming requires a background in designing with HDL languages. Furthermore an implementation is specific to a certain algorithm and cannot be used for any other algorithm even if it is slightly different. So if a new algorithm for a particular task is found then a part of the design process has to be redone. Also designing for FPGAs is a computationally intensive task as the whole design after simulation has to be synthesized and then placed and routed (P&R) for a particular FPGA every time the design changes slightly. This process of mapping the design can take hours or even days to compute for large designs. In recent years developments in High Level Synthesis (HLS) and OpenCL have made the whole process of designing for FPGAs an easier task. But this solution is not without problems either as the algorithm has to still be implemented for a specific FPGA device. A solution to the FPGA synthesis and P&R problem has recently been proposed with the name of FPGA Overlay Architectures. The core concept of this idea to abstract the FPGA create a virtual FPGA on top of the underlaying physical one in order to help with configuration and compile time. In this thesis, we investigate available alternative overlay architectures and select the most appropriate architecture for our analysis. We extended the selected architecture to be deployed on alternative FPGA hardware and to work in a shared CPU/FPGA system. Then, we implemented a number benchmarks to evaluate various aspects of system performance. Our results show that our architecture can be reconfigured in only 11.9us, as compared to seconds for full FPGA reconfiguration. However, the overlay architecture uses 10.5x more LUTs and causes a drop in frequency of about 30% for the chosen architecture. For future work, there is room to improve these results by optimizing the interconnect network of the device.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2017-12

Dedicated to my parents and my brother

Contents

List of Figures **vii**

Acknowledgements **ix**

1 Introduction **1**

1.1 Context 1
1.2 Proposed overlays solutions 2
1.3 Our use case 3
1.4 Research question 3
1.5 Approach and goal 3
1.6 Contribution 3
1.7 Thesis outline 4

2 Background & related work **5**

2.1 Principle of operation 5
2.2 Related work 7
2.2.1 IF and DySER 8
2.2.2 DeCO 8
2.2.3 Chosen architecture 9
2.3 DySER 9
2.3.1 DySER core 9
2.3.2 DySER input/output bridge 11
2.4 Platform 11
2.4.1 About ZYNQ 11
2.4.2 ZYNQ CPU 12
2.4.3 ZYNQ FPGA 12

3 Design **15**

3.1 Adaptation to the FPGA 15
3.1.1 Functional units 15
3.1.2 Dual operation functional units 16
3.1.3 Switch modification 17
3.1.4 Grid size 17
3.2 Integration 18
3.3 Connection with the bus 20
3.4 Software integration and using the overlay 21
3.5 Area usage 24
3.5.1 Whole peripheral usage 24
3.5.2 DySER area usage 24

3.5.3 Input/Output bridge 24
3.5.4 Core area usage 26
3.5.5 Tile and switch area usage 26
3.5.6 CDMA and AXI interfaces usage 26
3.6 Scaling for a bigger device 27
3.7 Frequency and power usage 27

4 Test setup **29**
4.1 Testing methodology 29
4.2 Test kernels 29
4.3 Example of mapping a kernel 30

5 Results & discussion **33**
5.1 Performance measurements 33
5.1.1 Latency 33
5.1.2 Throughput 34
5.2 Speedup 36
5.3 Reconfiguration 40
5.4 Area & Frequency Overhead 41

6 Conclusion and future work **43**
6.1 Conclusions 43
6.2 Future work 44

Bibliography **48**

List of Figures

2.1 Data flow graph produced by the C-code kernel above [1] 6
2.2 DFG of the kernel mapped to an overlay architecture [1] 6
2.3 DFG of the slightly modified kernel 7
2.4 Comparison between the two architectures 8
2.5 Difference when mapping a kernel 9
2.6 Configuration Registers form a large shift register in configuration mode 10
2.7 High Level Overview of Zynq platform[2] 12
2.8 Hard processor block integrated into Zynq-7000 series [2] 13
2.9 Available interconnects between PS-PL [3] 14

3.1 Modified functional unit 17
3.2 Switch architecture 18
3.3 Final high level architecture of the design 20
3.4 Block design of the peripheral containing the overlay 22
3.5 State machine for connection to the overlay 23
3.6 Image showing the FPGA area that is occupied by a) the core in yellow b) the input bridge in red c) the output bridge in pink 25
3.7 Power of the Design as reported by Vivado 28

4.1 State machine that use for testing purposes 30
4.2 DFG produced by the convolution kernel 31
4.3 Mapping of the Kernel on the Overlay 32

5.1 Throughput for each kernel implemented on the 4 versions of the design 34
5.2 Speed up of the matrix multiplication algorithm for different versions of the design 36
5.3 Speed up of the 1d-convolution algorithm for different versions of the design 37
5.4 Speed up of the kmeans algorithm for different versions of the design . . 37
5.5 Speed up of the stencil algorithm for different versions of the design . . 38
5.6 Time spent in each of the execution steps for the 64-bit bus overlay . . . 39
5.7 Time spent in each of the execution steps for the 32-bit bus overlay . . . 39

Acknowledgements

First of all, I would like to thank my supervisor Prof. Zaid Al-Ars for letting me work on this project but also giving me the freedom to work the way I wanted. His door was always when I had questions or requests.

I would also like to thank my daily supervisor Johan Peltenburg for helping me in understanding the topic and providing me with help and directions on different technical problems that I had.

Finally, I would like to thank all my friends here in Delft for making my stay and study here an easier task as well as helping me in difficult times.

Konstantinos Gkougkoulias
Delft, The Netherlands
December 1, 2017

Introduction 1

1.1 Context

In recent years due to the slowing down of Moores Law and Dennard Scaling new kinds of architectures using accelerators have been proposed to continue increasing performance and power efficiency of integrated circuits. These types of accelerators range from GPUs to FPGAs and even ASICs.

But with the cost of extra efficiency for the aforementioned accelerators comes the difficulty of programming them. The GPUs are the easiest to program compared to FPGAs and ASICs, and for parallel workloads they provide high performance at the cost of high power consumption.

ASIC offer the best performance and power consumption when compared to the other two solution but they lack programmability completely. Once a design has been implemented in ASIC it cannot be changed and this particular circuit will from now only be able to perform the specific functionality that it was designed for.

FPGAs on the other hand can sometimes provide the same performance as GPUs with better power efficiency and are reconfigurable. The downside of using FPGAs is the higher effort required in order to program them. Developing with them requires more time as well as a high degree of expertise and experience in the field, because someone needs to also be familiar with digital circuit design and computer architecture.

Once a design is verified for its correctness on a simulation level then it needs to be mapped onto a specific device. This process is called synthesis and place&route (P&R). These steps are too time consuming and can take hours or even days for larger designs, when many configurations need to be tested.

Even though FPGAs provide reconfigurability, once a design is mapped onto an FPGA then the FPGA can be used only to accelerate that particular algorithm, without paying a reconfiguration time to change its functionality. Moreover a design is targeted at a specific device and thus cannot be used with any other FPGA without passing the synthesis and P&R phase again, with some added modifications for the new device.

One more disadvantage is that if the algorithm changes slightly or a better algorithm is found, then some parts of the development process have to occur once more.

The lengthy compilation times are a result of the fact that the FPGAs are designed in order to be able to support many kinds of applications and are not specialized devices. So each time a design needs to be mapped onto an FPGA, millions of fine grain resources, such as LUTs, MUXs, etc, need to be programmed, which is the main cause of the long P&R times [4].

Overlay architectures on the other hand are build on top of the real FPGA structure, and are virtual structures that connect different kinds of abstracted functional units with an abstracted interconnection network.

This abstraction helps to hide much of the underlying complexity of the FPGA structure [5]. Instead now the programmer or the compiler can see only higher level coursed grained resources (such as adders, multipliers, square units, FFT units and so on). That makes the mapping of kernels on an FPGA an easier and faster task, as the algorithm has smaller design space to search and the configuration requires less bits to be transfered.

Also because overlays are virtual devices a configuration is portable from one device to another, as long as the overlay can be built on that device.

The only downside that comes with the use of overlays is the area increase when compared with a same implementation using purely HDLs. The programmability of the device is the cause behind this area increase.

1.2 Proposed overlays solutions

The advantages that overlay architectures bring can be used in one on the ways described below

(a) James Coole and Greg Stitt [6] propose that we can have a library of intermediate fabrics already implemented as bitfiles in the disk. Each of these intermediate fabrics supports a number of different but similar kernels. When a kernel is supported by the overlay it gets faster execution and reconfiguration time.

    If a normal FPGA structure would have been used then each time a new kernel arrived the whole FPGA would have to be reprogrammed, something that requires considerably more time. Using overlays this process has to be done only when a new type of kernel arrives, that is not supported by the current overlay laying on the FPGA.

(b) Tony Nowatzki et al [7] propose a new architectural model with the use of overlay Architectures. They connect a low power core with an overlay to accelerate execution of certain demanding parts of the code.

    The concept behind this idea, is the use of a general purpose CPU as load-store engine for the overlay architecture. CPU cores are also responsible for configuring the overlay architecture before the execution of each kernel.

Computationally intensive parts of the algorithms are going to be executed by the overlay while less demanding parts and parts that are not supported by its structure will be executed on the CPU.

1.3 Our use case

For our use case, overlay architectures are going to be evaluated for their performance in kernels that are used in big data [8] and scientific applications [9], but also we would like to know how fast they can be reconfigured to alter their functionality.

In the fields of big data analytics and scientific research, FPGAs have recently began seeing greater usage, for example in applications related to medical research, such as DNA sequencing [10] and medical imaging [11]. Users in these areas can take advantage of the low reconfiguration times to make better use of their FPGA devices.

1.4 Research question

Keeping the above mentioned points in mind, the main research questions that this thesis will try to answer are the following ones

(a) What kind of speedup can we expect when we connect an overlay architecture to a CPU, and which are the bottlenecks that limit potentially higher performance?

(b) How fast can we re-configure such an architecture so that a different kernel will be able to run on the same fabric without having to reconfigure the whole FPGA?

(c) What is the overhead for using such an architecture in terms of area and power?

1.5 Approach and goal

Recently there have been certain systems, like the ZYNQ platform by Xilinx, that enable easier integration of FPGAs with a general purpose processor. These systems provide a hard processor and an FPGA fabric on the same die. The two can communicate with low latency via the different interfaces available between the ARM core and the FPGA.

Therefore the goal of this thesis is to find such an architecture and try to integrate it with the ARM CPU available on this system, so that it can make use of the overlay as an accelerator for different kernels.

1.6 Contribution

In this section the contribution and the work that has been done on this thesis will be summarized.

- Carried out literature review for the current state of the overlay architectures.
- Analyzed the alternatives and choose such an architecture that can be integrated into an FPGA + CPU platform.
- Evaluated the chosen architecture and how it can be programmed to execute kernels.
- Modified some parts of the functional units in order to execute FP operations on Xilinx FPGAs. Also reduced the area of the switch, achieving a 25% reduction while retaining the desired functionality.
- Connected the architecture with the ARM CPU using the AXI interfaces and evaluated the performance by mapping kernels for scientific applications.
- Showed that such an architecture can be integrated to the current Xilinx ZYNQ platform and confirmed the advantages of overlay architectures, such as fast reconfiguration time. Also described the shortcomings of the current architecture.

1.7 Thesis outline

The thesis is divided into the following chapters

Chapter 2: This chapter will discuss the work already done by other authors in this field, and one of the available options will be chosen for implementation. The chosen architecture will also be presented in a bit more detail.

Chapter 3: In this chapter we will first discuss the architecture of the functional units and the switches of the system as well as the modifications that were made to them in order to better fit the chosen platform. We will also go into detail on how the design was integrated to the platform.

Chapter 4: The benchmarks that were used and the methodology for their evaluation will be presented in this chapter. An example of mapping a kernel will also be presented step by step.

Chapter 5: In this chapter, the performance results of the selected benchmarks will be presented. Results concerning reconfiguration time and area overhead will also be shown.

Chapter 6: At the final chapter a summation of the work that was done on this thesis along with guidelines for future work are going to be presented.

Background & related work 2

Overlay architectures have emerged in recent years as a response to the increasingly high times required to build a design on an FPGA. This includes the large amounts of time required for synthesizing and placing and routing the design as well as the complexity that comes from designing in very low level HDL languages.

In principle all overlay architectures try to hide the complexity of the underlying structure of an FPGA to the designer or the compiler. This means they do not have to worry about specific details of the underlying structure of the device and see it as an abstract entity, like the way Java runs on a Java virtual machine on top of a real processor. This greatly benefits compilation and reconfiguration times, due to the fact the compiler has less things to compute and less bits are required to program the device. But as with anything virtual this comes at the cost of higher area usage.

Another important characteristic of overlay architectures is the fact that they are programmable like a normal FPGA is. But what makes them more appealing that an FPGA in this aspect, is that they require far less time to be configured. In a normal FPGA, we have to configure the whole device every time we need to compute something different. In an overlay architecture the configuration is much faster because only a small number of things needs to be reconfigured.

This makes it possible to run different kernels on a device because we only have to pay a small price for the reconfiguration. So an overlay can accelerate different types of kernels with only a small time penalty between them, while a normal FPGA can only accelerate one kernel before having to reconfigure the whole design something that takes considerably more time.

2.1 Principle of operation

All of the available overlay architectures rely on the same principle. They try to "draw" a data flow graph (DFG) of a kernel on the structure of the overlay.

Lets suppose for example that we have the following piece of C code that happens to be the kernel of an algorithm.

Listing 2.1: Kernel of an algorithm in C

```
int foo(int a, int b, int c, int d){
        return ((a+b)*(c+d))-d;
}
```

The kernel above produces the following data flow graph (fig. 2.1)

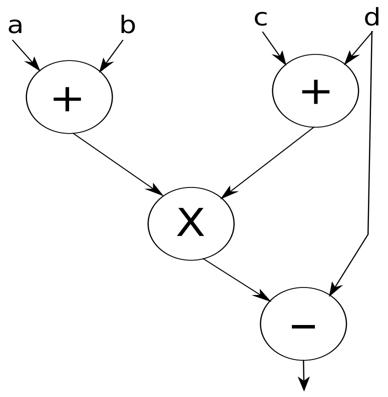


Figure 2.1: Data flow graph produced by the C-code kernel above [1]

Suppose that we have a 2x2 grid of the functional units, that each of them can implement operations like addition and multiplication, and switches that are responsible for transferring data between the functional units. Then this data flow can be mapped on the overlay architecture as seen in the following figure (fig. 2.2) (a rhombus represents switches while a squares represents functional units).

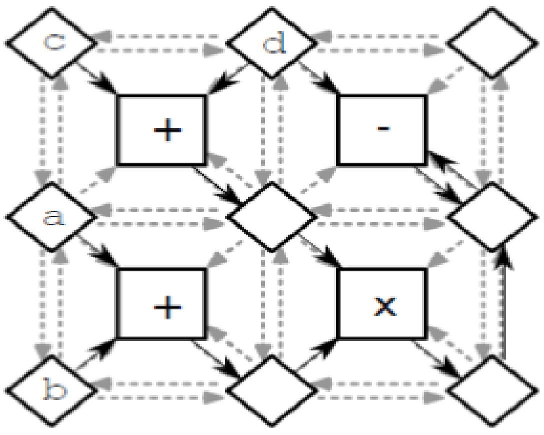


Figure 2.2: DFG of the kernel mapped to an overlay architecture [1]

Now imagine that a new kernel arrives, that is slightly changed similar the one that can be seen in the following code section

Listing 2.2: New modified kernel

```
int foo(int a, int b, int c, int d){
        return ((a-b)*(c+d))-d;
}
```

This new kernel will require the DFG shown in the next picture (fig. 2.3).

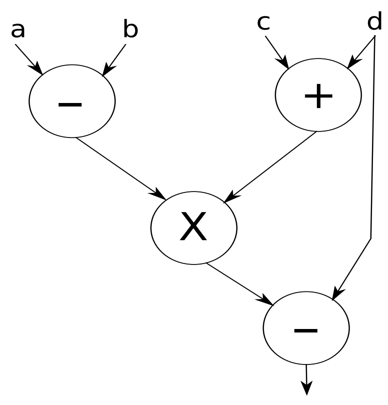


Figure 2.3: DFG of the slightly modified kernel

Using the property of overlay architectures, we can quickly reconfigure the fabric in order to support the new kernel. In that case a new programming sequence needs to be send that will change the functionality of the bottom left functional unit shown in fig. 2.2 from an addition to a subtraction.

Functional units (FUs) can be anything as simple as an adder or a multiplier and as complex as an FFT unit or even a whole processor. But common FUs (adders, multipliers, sqrt) are considered first as they can be used for almost every kernel.

While the FUs are fixed after generation their interconnection is configurable, so that they can support different number of kernels.

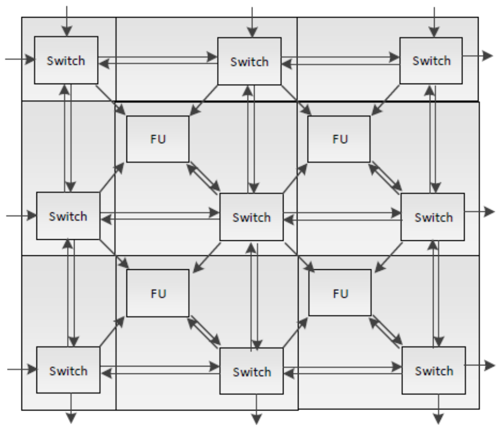
2.2 Related work

In the past there have been a number of different overlay architectures that rely on the above principle. The most researched ones are the following

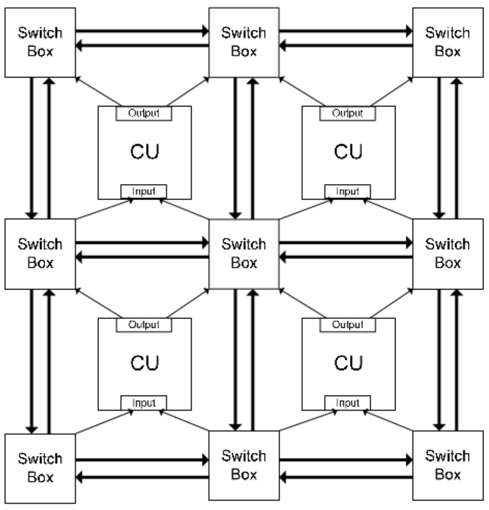
(a) Intermediate Fabrics by James Coole and Greg Stitt [4, 5, 6]

(b) DySER by Vertical Research Group of the University of Wisconsin-Madison [12, 1]

(c) A number of different sorts of designs by Abhishek Kumar Jain et al [13, 14, 15].



(a) DySER high level architecture [16]



(b) IF high level architecture

Figure 2.4: Comparison between the two architectures

The first two are quite similar and they both rely on the same island-style overlay but with different interconnection schemes. In the third option Abhishek Kumar Jain came up with a different number of overlays similar to the first two ones, but also with one with different architecture that will also be presented.

2.2.1 IF and DySER

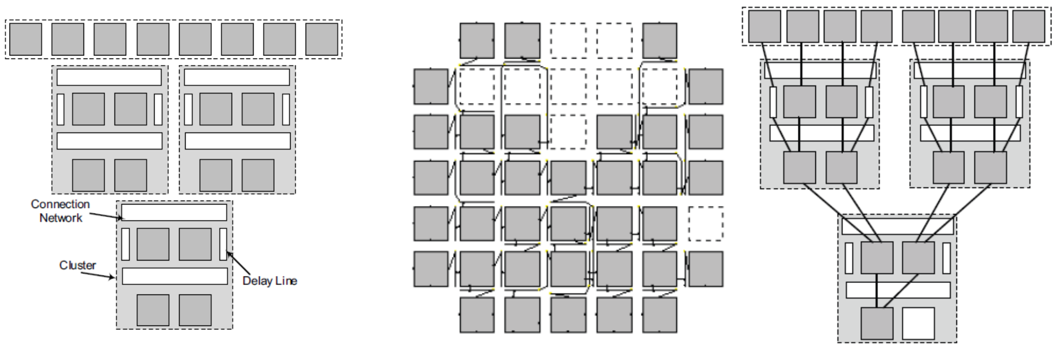
Intermediate Fabrics (IF) and DySER rely on the same high level architecture of island style overlays. This is similar to the way that an FPGA is organized, where DSPs and LUTs are connected via a configurable interconnect network.

In the figure above (fig. 2.4) a comparison between the two architectures can be seen. The similarities between them are quite obvious. The difference is that DySER can accept inputs from 4 different directions while IF only from 2. The output of DySER is always directed at the SE of the FU while on the IF there are two options.

2.2.2 DeCO

Abhishek Kumar Jain et al have researched different overlay architectures most of which are similar to the ones already described. The one that is different is called DeCo [15] and does not use the island style architectural model.

So he proposed a different architecture he calls linear interconnect architecture. This is a cone shaped architecture, similar to a reduction tree operation. The data can flow only in one way thus reducing the interconnection complexity, but which also reduces the generality of the architecture.



(a) Deco high level architecture

(b) Difference when mapping a kernel

Figure 2.5: Difference when mapping a kernel

The figure above fig. 2.4 shows how such an architecture looks like and the difference with an island style, when mapping a kernel.

2.2.3 Chosen architecture

The architecture that was chosen to be implemented is DySER, since it is the only design available open source and does not have principal differences with the other architectures. The design is available at this website [17].

In the following section a more detailed description of the DySER architecture will be presented.

2.3 DySER

2.3.1 DySER core

On the highest level the DySER core consists of two components, the edge fabric and the tile fabric. Edge fabric lies on the left and top edge of the core. The rest of the design is the tile fabric.

A design is characterized by its grid size and is of square size, for example 5x5 or 6x6. In general a NxN grid size contains N^2 functional units and $N^2 + 2N + 1$ switches.

The Edge Fabric is made out of edge tiles, that are just a single switch, and are responsible for directing the data that is coming from the bus. The tile fabric is composed out of tiles. Each tile contains a functional unit alongside a switch.

Both tiles contain a register that is responsible for programming the data flow and the functionality of the functional unit when present. In order to program the overlay, when in configuration mode the switches form a large shift register as it can be seen in the picture below (fig. 2.6), which is essentially a daisy chain configuration.

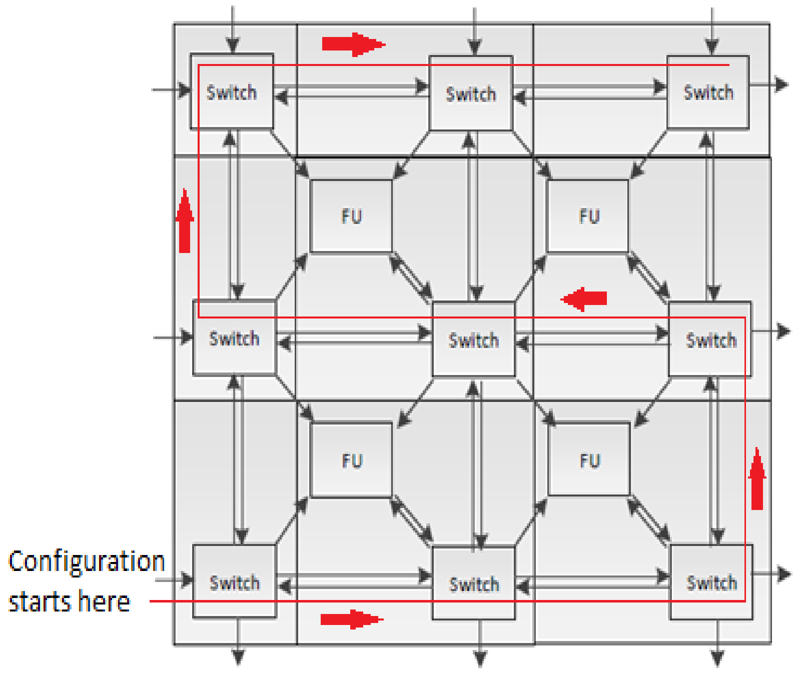


Figure 2.6: Configuration Registers form a large shift register in configuration mode

The way that functional units work is that they have 4 inputs on NW, NE, SW and SE. According to the programming of the FU two of these 4 inputs are selected and the desired function is computed. Then the result is always outputted to the SE of the functional unit and goes into the neighboring switch. Each FU contains one or more computational resources and logic that is responsible for the synchronization.

Switches have 5 inputs (in the N, E, S, W and NW) and 8 outputs to every direction. That requires a 5to1 multiplexer at every output as well as a state machine for synchronization at each output.

To allow latency imbalances into the calculation of the DFG each FU also implements a state machine for the synchronization. This state machines implements a credit-based flow control protocol using a forward signal (valid) and a backward signal (credit) [18]. Functional units perform operations only when all inputs are valid and data is forward only when the credit signal is asserted. Functional units and switches send credits only when they can accept new data. This forms a pipelined execution model between different FUs and takes cares of the latency imbalances between the data arriving at the ports.

2.3.2 DySER input/output bridge

The core has a large number of data inputs, that each is made out of 32 bits. This means that the CPU cannot be directly connected to each of the those inputs because that wide buses do not exist in the targeted system.

So an input bridge must be created in order to connect a bus to the core. This looks like a multiplexer but with added FIFOs at the end of each output. The FIFOs help in buffering data to the core while the core is occupied doing calculations.

The same holds true for the output of the core as the overlay has a large number of outputs, each also made out of 32 bits. A similar device needs to be constructed to connect these outputs to the bus. At the output of such a device there are also FIFOs because many results can be produced before the CPU reads them or they are written to RAM.

2.4 Platform

In order to be able to realize an architecture like that, a platform that provides both a general purpose processor and an FPGA fabric as well as the interconnect that enables the communication of the two, is required .

The Xilinx Platform ZYNQ is an ideal candidate as it includes a hard-core general purpose processor along with an FPGA fabric and interconnections, all on the same die. This enables fast communication between the two, and also makes their integration an easier task for the designer.

2.4.1 About ZYNQ

Zynq-7000 [19] is a platform developed by Xilinx that integrates a number of ARM v9 CPU cores along with Artix-7 or Kintex-7 FPGA fabric. Those two are built of the same piece of silicon and a number of AXI interfaces are used to connect them, as can be seen in the following image (fig. 2.7). The particular board that is going to be used, is called PYNQ and contains the XC7Z020-1CLG400C chip with 512 MB of RAM. This chips offers the resources seen in the following table (table 2.1).

	Available
LUTs	53200
Registers	106400
DSPs	220

Table 2.1: Resources available at the FPGA

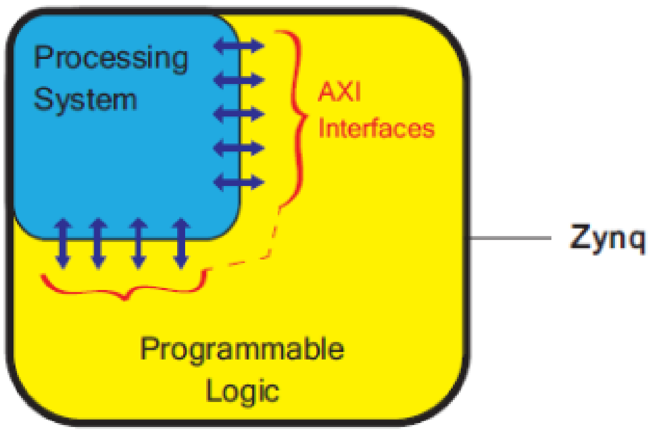


Figure 2.7: High Level Overview of Zynq platform[2]

This tight integration enables software and hardware to be partitioned into different development phases and then integrated.

2.4.2 ZYNQ CPU

In the past most FPGA devices did not contain a hard general purpose processor so they were unable to execute general purpose code in high level programming languages. To overcome this it was possible to build a soft processor on the FPGA fabric, a processor that was build using the FPGA LUTs and BRAMs. For example Xilinx had a processor IP called Microblaze that someone could use with their FPGAs.

The disadvantage of this approach was the considerably lower clock speeds and the large area overhead when compared with a hard processor block.

In the following image (fig. 2.8) the overview of the ARM A9 CPU used in Zynq-7000 can be seen. It can be observed that it not only provides a CPU core but also various interfaces such as USB, SD, Ethernet, UART etc. that can be used to connect it with peripheral devices.

In Xilinx terminology this part of the system is called PS (Processing System) and will be referred this way from now on. Except the peripheral devices interfaces, the thing that is most interesting to us is the presence of different AXI interfaces that connect it with the FPGA fabric.

2.4.3 ZYNQ FPGA

As we have already mentioned the are several AXI4 interfaces that connect the FPGA fabric with the ARM CPU. AXI4 [20] is a standard developed by ARM and is the default bus used by Xilinx for all of its IPs, as well as a number of other vendors of embedded systems. AXI4 comes in different versions that are the following

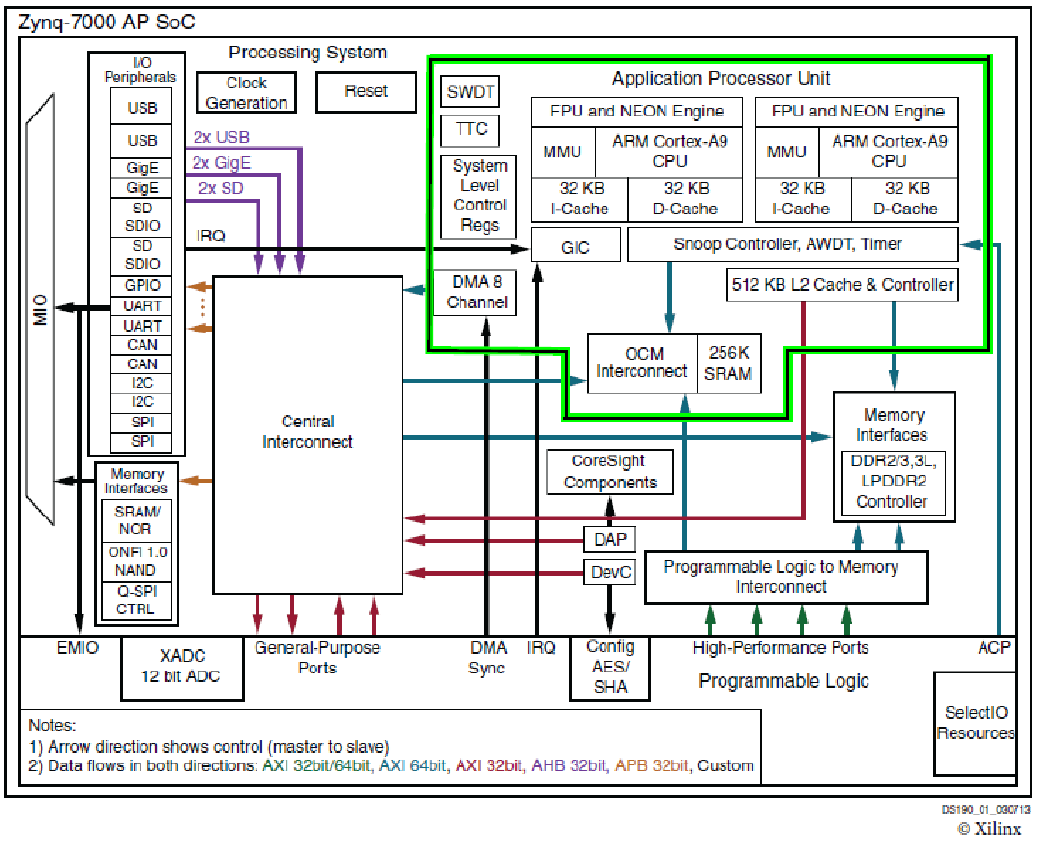


Figure 2.8: Hard processor block integrated into Zynq-7000 series [2]

(a) **AXI4:** This is the full implementation of the protocol that is used for high performance memory mapped applications. It can support burst of up to 256 data transfers with a single address phase.

(b) **AXI4-Lite:** This is the light-version of the interface that supports memory mapped transactions. It supports only single data transfer and is suitable only for writing and reading to control and status registers.

(c) **AXI4-Stream:** This interface removes the requirement for an address phase in the transactions process and can support unlimited data transfer bursts. This interface is not memory mapped and it can be used only for data streaming.

In the following figure (fig. 2.9), the different interfaces that are available for the interconnection between PS and PL (Programmable Logic) are presented in greater detail.

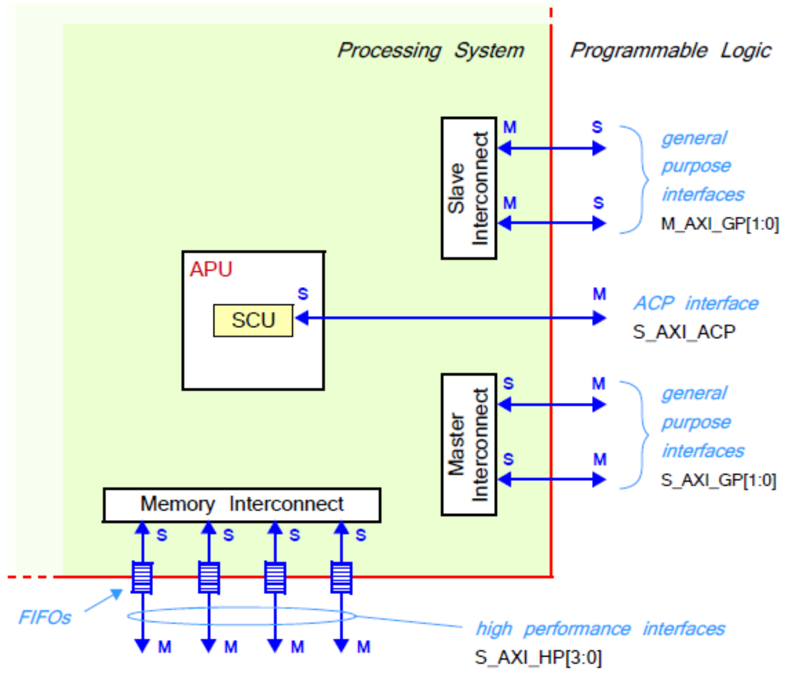


Figure 2.9: Available interconnects between PS-PL [3]

From the above interfaces the general purpose ones implement the AXI4-Lite protocol, so they provide lower bandwidth and no burst support, making them suitable only for controlling registers.

The HP Ports (High Performance) implement the AXI4 or AXI4-Stream protocol, and can support bursts of 256 beats for the AXI4 protocol case. There are connected straight to DDR so they are suitable for transferring large amount of data when high performance is required.

Design 3

In this chapter there will be a discussion over the changes that were made to DySER to be more suitable for implementation on an FPGA [16]. Also we will discuss how we were able to integrate the overlay onto the ZYNQ platform and how it communicates with the CPU.

3.1 Adaptation to the FPGA

3.1.1 Functional units

Dyser was not initially targeted for FPGA implementation but was a general design. This means that certain aspects of the design are not that suitable for FPGA mapping.

The most important thing that falls under this category is the functional units that are used in DySER. They are generic implementations of modules that implement FP adders, multipliers and other floating point operations. While implementation on ASIC maybe optimal or acceptable, such implementations on FPGA suffers from low frequency and high LUT consumption.

Also using such implementations we are not taking full advantage of the FPGA resources that are dedicated for this purpose, the DSP blocks. Thus using the DSP blocks to implement them will improve area usage as well as frequency [16].

DSP blocks can support only integer operations out of the box. In order to make them usable for floating point operations we need to extent their functionality. Luckily, Xilinx offers floating point units [21] that are optimized for each of its devices. These combine a number of DSP blocks and surrounding LUTs to add the required functionality to support floating point operations.

They can be generated through Vivado and can implement any desired FP operation. More parameters are also available like the desired latency, the type of interface etc. These blocks are generate as encrypted files and their functionality can not be altered. In essence they act like black boxes that receive inputs and after a certain number of cycles produce the desired result.

As already discussed Vivado Software can generate versions of the FP units with different amounts of latency from 1 till some number (10-12 usually), depending on the design. Initially the FP units with latency for 8 and 11 cycles were generated for multiplication and addition/subtraction respectively. But it was found that they were not producing the right result, as there was a problem with flow-control protocol for the communication between FUs and switches. The largest number for latency that was not producing a problem was found to be 4.

Less latency means that the module cannot reach that high of a frequency. The design was implemented with both functional units that had the most pipeline stages as well as with units that had 4 pipeline stages and no difference was found in terms of frequency. This is due to the fact that the critical path in the design lies in AXI interfaces and not on the computational units them selfs.

3.1.2 Dual operation functional units

In the reference HDL each functional unit consisted of one FP functional unit that could only implement one operation. For example each functional unit was composed out of an adder/subtractor or a multiplier or a divider along with the auxiliary logic required for flow control.

Initially every general functional unit was replaced with the generated unit provided by Vivado. Each generated functional unit uses then 2 DSP blocks so the whole overlay would use 50 DSP, out of the 220 available. That is a very poor usage of the FPGA resources as only 25% of the computational blocks were used.

Also since each functional unit was able to perform either addition/subtraction or multiplication a problem arose with the mapping of kernels. On a 5x5 grid it was unable to map more than one kernel of the chosen kernels, that were the sum of absolute differences, matrix multiplication/convolution and stencil. This meant that for each kernel a specific overlay with different functional units had to be constructed, something that defeats the purpose of overlays because we would have to program the FPGA every time a new kernel would appear.

To overcome this problem 2 Floating point modules were included in each functional unit, one for FP addition/subtraction and one for FP multiplication. At the time of programming the overlay, the function can be chosen from one of the three available ones. This allowed all kernels to be mapped onto a single 5x5 overlay fabric and also made the routing of kernels a much easier process.

It also improved the usage of the DSP blocks from 25% to 50%. One could expect that this would have also increased the LUT consumption by lot, but that change increased the area only by 5%. This reveals the fact that the switch network is probably the part of the design that requires the most area.

In the following figure (fig. 3.1) the modified functional unit can be seen. The input MUXs can be programmed to select 2 out of the 4 available the inputs to be the inputs of the computational resources. Also the functionality of the adder can be programmed to perform either an addition or a subtraction. Finally one of the two results is selected at the output.

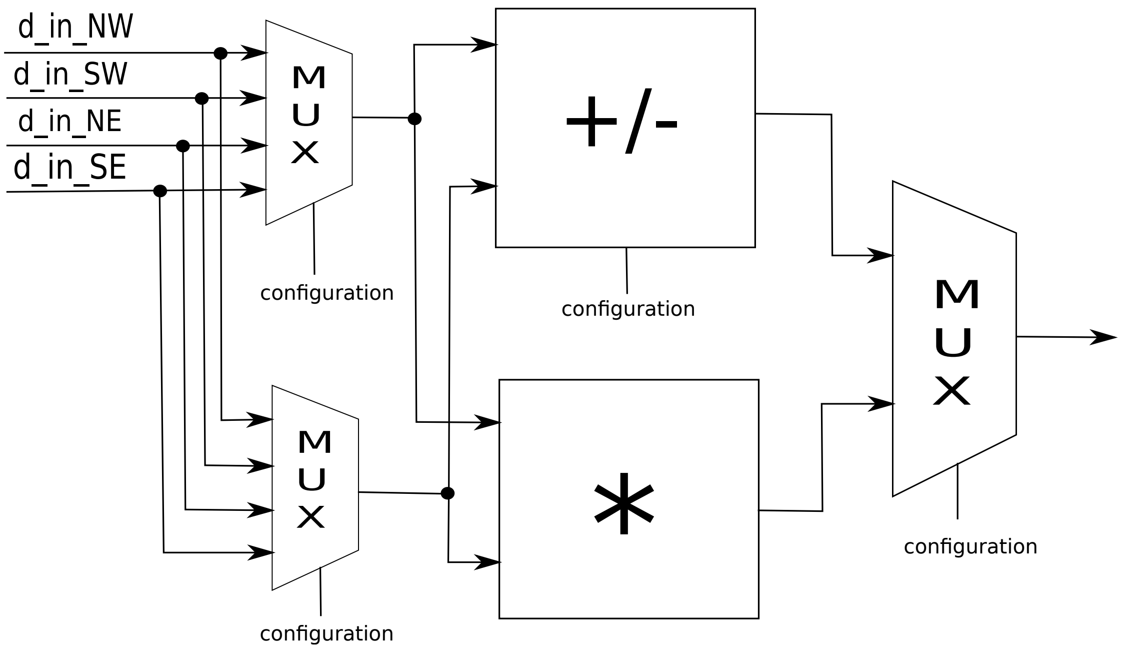


Figure 3.1: Modified functional unit

3.1.3 Switch modification

The reference design of the switch had two parallel multiplexers, that each was selecting one of the inputs, and then a third multiplexer that was selecting one of the these outputs. But the way we programmed the switches, was making use only of the results of the first switch and never from the other. That meant that the second multiplexer could be completely removed without affecting the design desired functionality, in the way that we intend to use it.

This gave a reduction in area of the whole design of about 10% for a grid size of 5x5 and about a 20% reduction in the area of the switch. If a functionality of the design called predicate, needs to be used then the second multiplexer should also be included.

In the following figure (fig. 3.2) the design of a switch can be seen. As it can be observed at every of the 8 outputs, there is a multiplexer that can be configured to select one of the five inputs entering the switch. This way the data coming from the overlay inputs or the functional unit outputs can be directed to the fabric.

3.1.4 Grid size

The reference design offered by the creators of Dyser had a grid size of 8x8, offering 64 Functional Units with 81 switches. Our target device was a Zynq-7020 SoC, which contains 53K LUTs and 220 DSPs. As we target single precision algorithms for

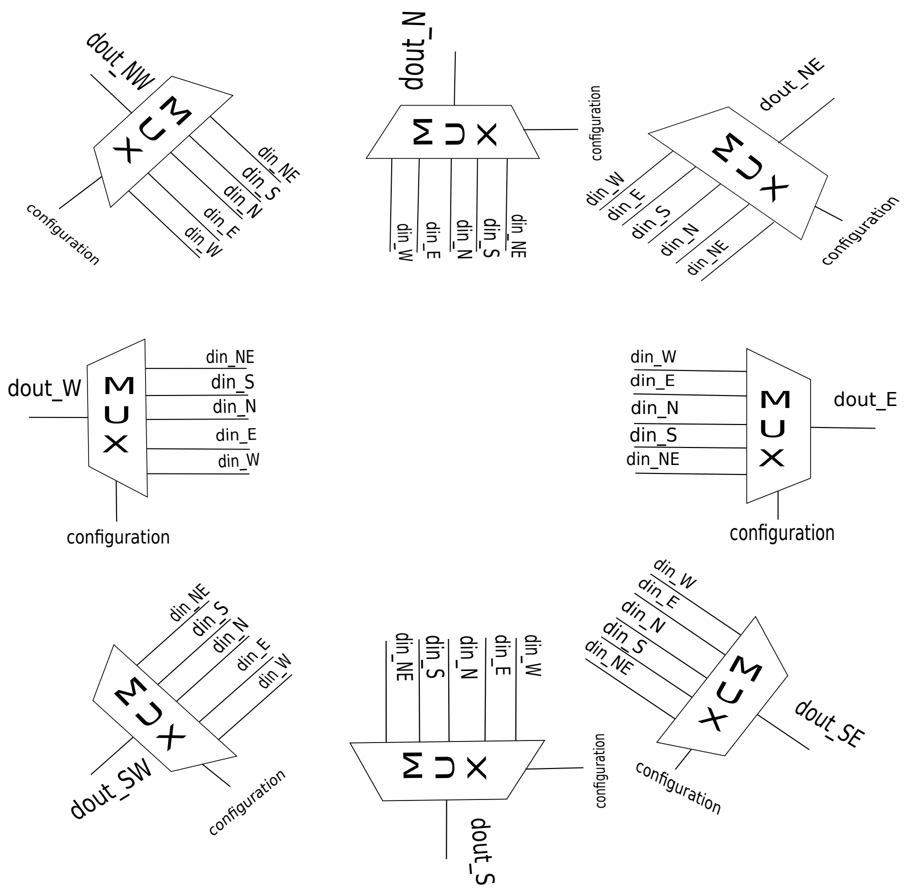


Figure 3.2: Switch architecture

implementation, the size of the FUs and the connections required need to be 32 bits.

This overlay size was unable to fit in the device, so the size of the grid had to be made smaller. The initial design was taken as a base and a NxN part was removed from it to create the new size. After that some connections between the components had to be adjusted so that they follow the structure of the 8x8 grid.

After a few tries it was found that the largest size that can comfortably fit on the FPGA is 5x5. The resource that is limiting the use of bigger grid sizes is the number of LUTs that are available on the particular device.

3.2 Integration

The overlay is only one part of the design, as an overlay needs to communicate with the CPU and the main memory. Without a CPU the overlay cannot be programmed and also cannot know which data are required each time for a particular kernel calculation.

As we have already discussed in the previous chapter, the are many ways that the CPU can be connected to the FPGA fabric. Because the peripheral is memory mapped we could use one of the following methods

(a) **General Purpose Port with AXI4-Lite protocol**: With this option CPU is the master and the peripheral the slave. This means that the CPU can initiate transactions and the peripheral is responsible for responding to it.

    AXI-4 Lite interface is capable only of 32-bit transactions and no burst support. Therefore is suitable only for reading and writing control register and not sending large amount of data to the peripheral. Using AXI4-Lite for data transfer could require an address phase for each word send to overlay and could also keep the CPU busy with doing that.

(b) **High performance port with AXI4 protocol**: A better option is to use a general purpose interface as well as a high performance one. The general purpose is used only for register reading and writing and the high performance interface for sending data to the overlay.

    While in the general purpose interface CPU can be the master and send data from the GP interface, it cannot be the master on the high performance port. This means that the CPU cannot directly initiate a transaction on this port as only a device on the PL can be the master. In order to overcome this problem a DMA engine had to be used.

    A DMA (called Central DMA (CDMA) [22] in Xilinx terms) is an IP that can be built on the PL part of the ZYNQ and can act as a master to the high performance ports. It can free up the CPU from the task of sending large amounts of data to the peripheral. The CPU is connected to the CDMA via the GP port with it being the master. This way it programs the CDMA to carry out the transaction in its place.

    The CPU sends to the CDMA the starting source address of the data we want to transfer, the starting destination address and the number of bytes that needs to be transfered. Then the CPU is free to do other tasks while the CDMA is responsible for the transfer.

    Finally, when the overlay finishes a calculation the result needs to be transfered to the main memory. This is done by the CPU that polls the status register of the peripheral to know when a result is ready, then reads the results and writes it to the main memory.

    A more optimized way would have been to make use of interrupts so when all calculations are finished, only then the CPU commands a second CDMA engine to start transferring data to the main memory.

In the following two figures (fig. 3.3,fig. 3.4), the way that all devices described above are connected can be seen, as well how the peripheral is internally organized. This is the final design that makes use of both the GP and HP ports of the device.

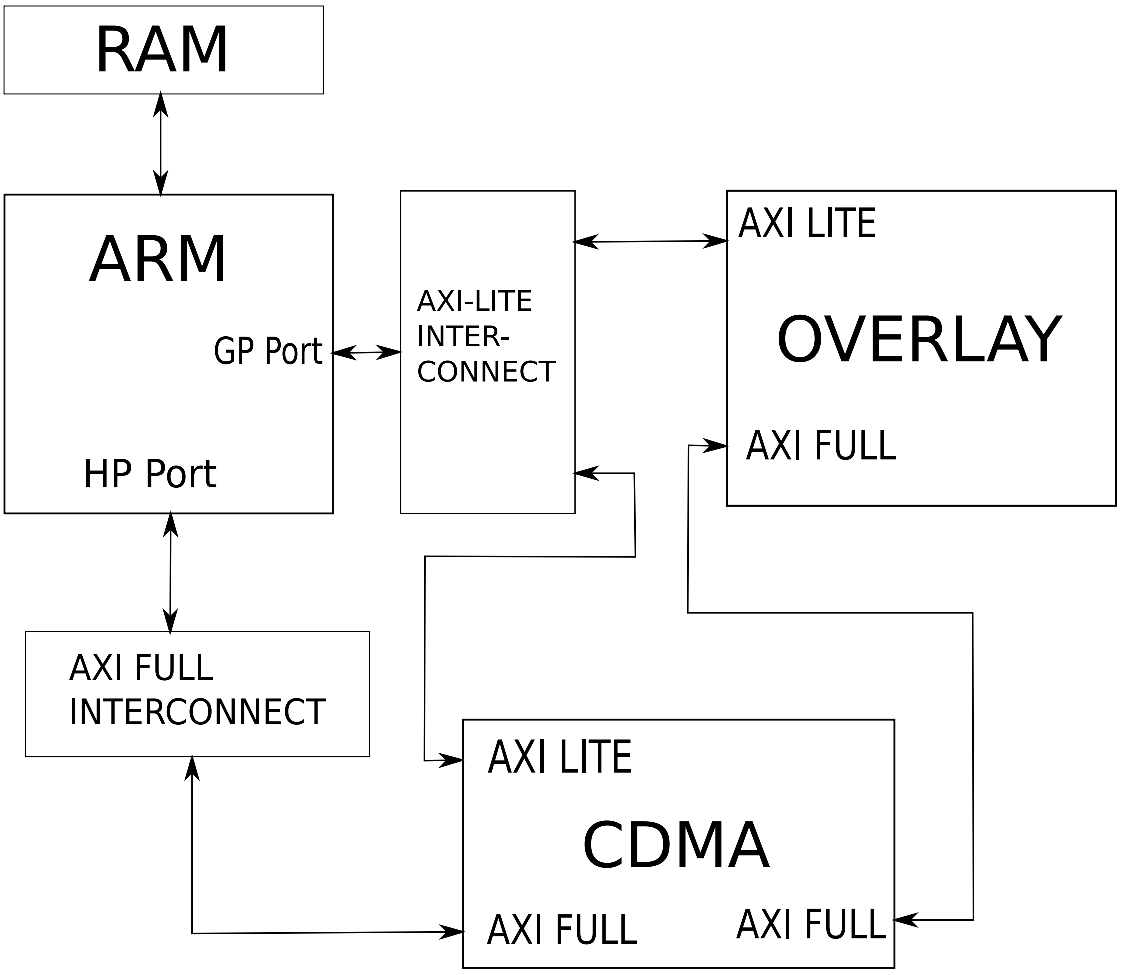


Figure 3.3: Final high level architecture of the design

3.3 Connection with the bus

In order to connect the bus with the Overlay we need to the logic that implements the desired bus protocol. With Xilinx Vivado a AXI4 peripheral can be generated that includes the required logic for that protocol. But this alone is not enough, as we also need a state machine to control the data coming from the bus. This state machine will be responsible for controlling the behavior of overlay after receiving commands from the CPU.

The states machines follows a control and status register scheme to communicate with the CPU. In the control register the CPU writes the commands to control the peripheral. These command tell to the peripheral to go to configuration mode or instructs the output bridge to expect results in certain ports etc.

There is also a status register that the overlay sets to 1 when the FIFOs are not empty, which means that there is result ready. The CPU knows then that data in FIFOs are valid and starts reading them. When the FIFOs are empty the status register goes to 0 and the CPU stops reading the FIFOs.

Initially the overlay is in the idle state, and when receives a command it goes to CONF state. In this state the overlay is in configuration mode and the content of the configuration registers can change.

When the CPU sends all the required configuration data to the overlay, it issues a command to change the state of the overlay. Along with that it sends the port or ports, where the results are expected to exit the overlay. Also the ports for receiving data are enabled and the overlay can start receiving data for calculations.

An image of the state machine described (fig. 3.5) as well as a block diagram of the peripheral (fig. 3.4) can be seen below.

3.4 Software integration and using the overlay

After finishing with the hardware implementation and integration, the software side that runs on the CPU was needed to be developed.

We took a couple of kernel implementations from a benchmark suite and two more from other sources and tried to modify them to use the accelerator. In certain parts of the code we inserted some instructions that make use of the overlay. In the following section the way that an algorithm is scheduled for execution on the overlay will be presented.

The configurations of the overlay are saved into an array. These are the sequences that will program the overlay for the desired kernel. For the particular case of a 5x5 overlay this sequence is 1568 bits long. The CPU initially writes the control register of the overlay to instruct it to enter in configuration mode. After that it starts sending the configuration sequence to it using the CDMA.

When this process is done, it sends an instruction to the overlay to move into the calculation state. Then the algorithm start executing on the CPU, and when the kernel is reached it programs again the CDMA engine to start sending kernel data from a certain point in the main memory to the ports of the overlay that are used as inputs to the core. Then the CPU is freed from the task of sending data and starts polling the

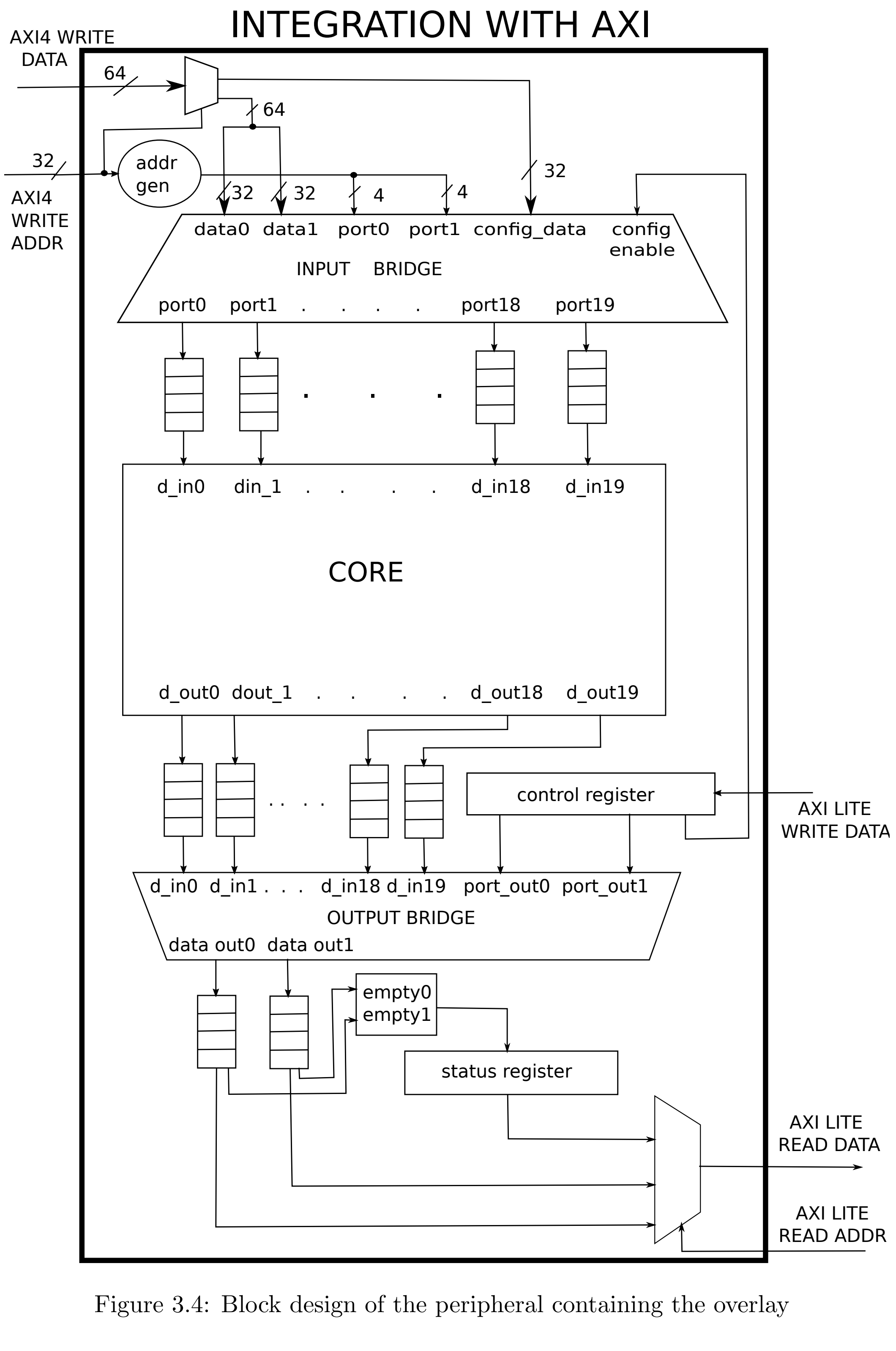


Figure 3.4: Block design of the peripheral containing the overlay

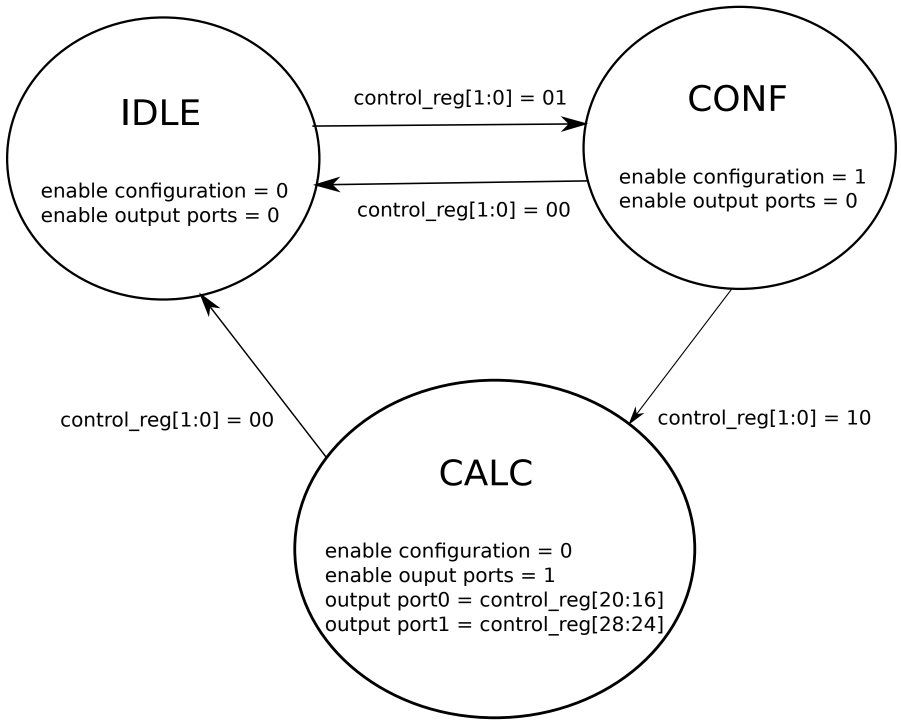


Figure 3.5: State machine for connection to the overlay

status register of the overlay to read the results when available.

At each input of the overlay there are 4-wide FIFOs, so that the CPU can send up to 4 data vectors for calculation. So after instructing the CDMA to do the job in its place, it starts reading the output and needs to read the results for all the calculated data before proceeding to the execution of the next instances of the kernel.

This process continues until the calculation for all the kernels is completed, or the overlay needs to be programmed again for a different kernel.

One problem that was observed while creating the software component of the design is the memory coherency between cache and main memory. Since the CDMA engine is not contained in the CPU package, it is unaware if there are "dirty" data in the main memory that are still stucked in the cache. If that fact is ignored then wrong data are sent to the overlay.

To overcome this problem the data cache needs to be flushed for the particular region of interest and write the current data back to the main memory before initiating any CDMA transfer. This is a point that effects negatively the performance of the overlay and gives an advantage to the CPU that has the mechanisms to overcome it.

3.5 Area usage

In the following subsections a overview of the area usage for each part of the design will be presented.

3.5.1 Whole peripheral usage

First of all, the area that the whole peripheral occupies on the FPGA fabric is show. This includes everything that has to do with the DySER and the logic that is needed for it to be connected to the AXI4 buses, as well as the state machines that control the overlay and the data flow.

What this area measurement does not contain is the area that is been used by the AXI CDMA (the DMA engine that is responsible for transferring data from the main memory) and well as the various AXI interconnects that connect the modules on the FPGA with the CPU and the main memory.

	Used	Available	Percentage
LUTs	40357	53200	75.8%
Slices Regs	26918	106400	25.3%
Slices	11008	13300	82.7%
DSPs	100	220	45.5%

Table 3.1: Peripheral area

3.5.2 DySER area usage

This area measurement includes everything that is directly related to DySER, namely the input/output bridge and the core. It does not include the auxiliary logic that helps it communicate with the bus such as the AXI4 protocol implementation and various other states machines.

	Used	Available	Percentage
LUTs	39524	53200	74.3%
Slices Regs	25963	106400	24.4%
Slices	10807	13300	81.2%
DSPs	100	220	45.5%

Table 3.2: Overlay + Input/Output bridges area

3.5.3 Input/Output bridge

These two bridges help to connect the inputs of the core to the buses in order to send a receive data. They are essentially large multiplexers with 4-wide FIFOs connected at their outputs.

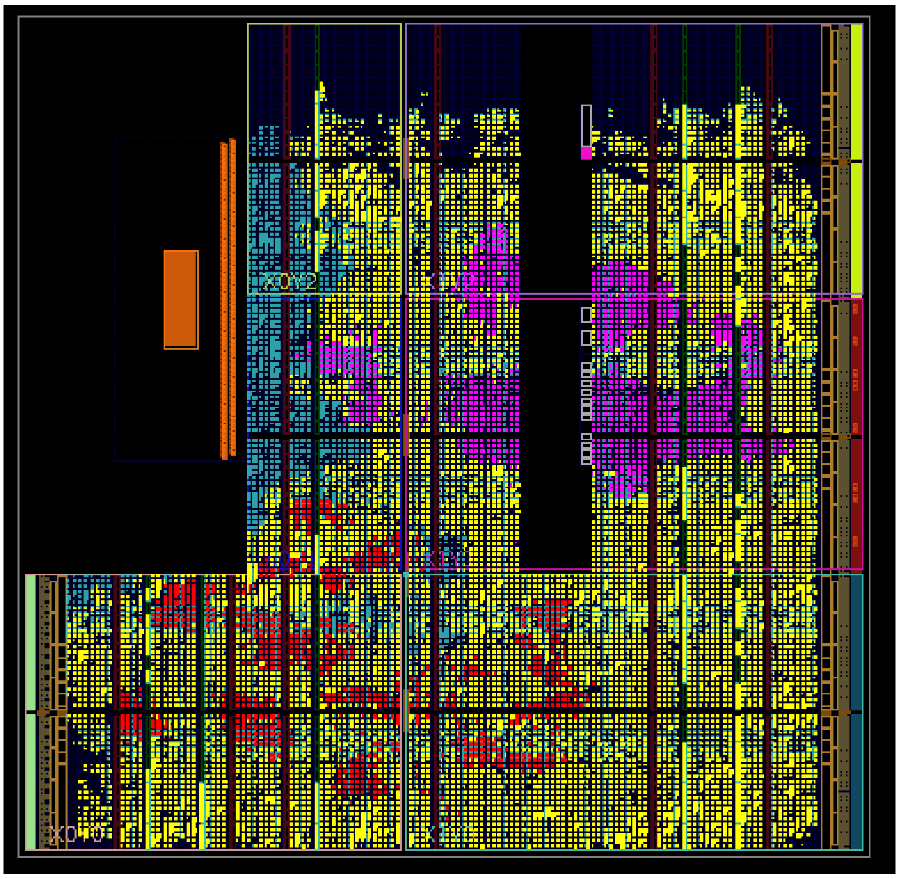


Figure 3.6: Image showing the FPGA area that is occupied by
a) the core in yellow b) the input bridge in red c) the output bridge in pink

	Used	Available	Percentage
LUTs	3365	53200	6.3%
Slices Regs	2640	106400	2.5%
Slices	1087	13300	8.2%
DSPs	0	220	0%

Table 3.3: Input bridge area

	Used	Available	Percentage
LUTs	3959	53200	7.4%
Slices Regs	2620	106400	2.5%
Slices	1097	13300	8.2%
DSPs	0	220	0%

Table 3.4: Output bridge area

3.5.4 Core area usage

In the core area only the logic that is related with the computation itself is included. These are the switches and functional units that help to map a DFG onto the overlay architecture.

	Used	Available	Percentage
LUTs	32200	53200	60.5%
Slices Regs	20703	106400	19.4%
Slices	9162	13300	68.9%
DSPs	100	220	45.5%

Table 3.5: Core area

3.5.5 Tile and switch area usage

Finally, we are going to take a look that the smallest building block of the architecture, the tile. The tile contains a functional unit and a switch.

When comparing the two, it is easily observed that the switch is consuming the largest part of the tile area, something that confirms our previous assumption. That means that an improvement in the switching network will have a great positive effect on the design.

	Used	Available	Percentage
LUTs	1121	53200	2.1%
Slices Regs	739	106400	0.7%
Slices	335	13300	2.5%
DSPs	4	220	1.8%

Table 3.6: Tile area

	Used	Available	Percentage
LUTs	758	53200	1.4%
Slices Regs	376	106400	0.3%
Slices	239	13300	1.7%
DSPs	0	220	0%

Table 3.7: Switch area

3.5.6 CDMA and AXI interfaces usage

Finally, the area usage for anything that is not included in the peripheral is going to be presented. This includes the CMDA engine and the logic that implements the various AXI interfaces.

	Used	Available	Percentage
LUTs	364	53200	0.7%
Slices Regs	757	106400	0.7%
Slices	144	13300	1%
DSPs	4	220	1.8%

Table 3.8: Functional unit area

	Used	Available	Percentage
LUTs	2023	53200	3.8%
Slices Regs	2563	106400	2.4%
Slices	863	13300	6.4%
DSPs	0	220	0%

Table 3.9: CDMA+AXI area usage

3.6 Scaling for a bigger device

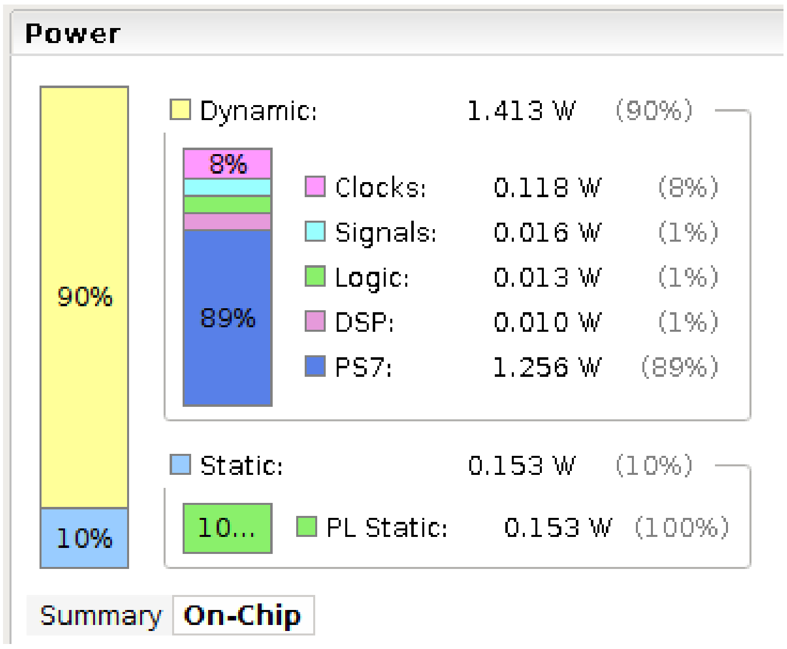
Additionally, we would like to see how big of an overlay can fit in a state of the art FPGA. The FPGA that was chosen was the XCVU9P [23], which offers 2.58 million LUTs which is the limiting resource in our case. When synthesizing the 5x5 overlay for this device, it occupied 4.33% of the total LUTs.

Because area grows quadratically when increasing the size of the overlay, extrapolating from this result a 20x20 overlay that is 4 times larger in each direction would require 16x times more LUTs. This means that it would occupy about 60.5% of the total device.

3.7 Frequency and power usage

Lastly, the achieved frequency and power usage of the design will be shown. When the design was first integrated, it had a frequency of 72 Mhz. The problem was with a couple of AXI signals and the interconnection between the control and the data part of the design due to the fact that the two are implemented in a different modules. Also the reference design had an asynchronous reset that needed to be changed. After all these improvements were implemented, the achieved frequency was 125 Mhz.

As for the power usage of the design this can be seen in the figure below (Figure 3.7). As it can be seen the whole chip consumes 1.413 Watts of power. From those 1.256 W are used by the the PS7 which is the processing system, and only 0.157 Watts from the logic in the FPGA fabric.



Power

Dynamic:	1.413 W	(90%)
Clocks:	0.118 W	(8%)
Signals:	0.016 W	(1%)
Logic:	0.013 W	(1%)
DSP:	0.010 W	(1%)
PS7:	1.256 W	(89%)
Static:	0.153 W	(10%)
PL Static:	0.153 W	(100%)

Figure 3.7: Power of the Design as reported by Vivado

Test setup 4

4.1 Testing methodology

In order to make an evaluation on the effectiveness of the overlay and to quantify the weak point that need to be improved, a number of data parallel kernels was chosen from the Parboil benchmark suite [24] together with some other kernels. This benchmark contains kernels that are used in scientific applications like Matrix-Matrix multiplication, stencil operations on Matrices etc.

The code for the CPU was used and adopted each time to be able to make use of the overlay in the way that was described in the previous chapter. So at the end two version of the benchmark will be available, one running just on the CPU and one running on the Overlay + CPU.

In order for the algorithms to work they need data, that are available on the main memory in both versions. The structure of data in memory is the same for both cases, so that they can be compared fairly.

The time it takes for the whole execution of the program will be measured in both implementation of the algorithms to indicate the speedup. But this will not show the full potential of the overlay due to the limited width of the bus or the communication overheads. However systems that are intended to be used in high performance applications, have the capabilities to overcome this problem as they provide way faster and larger buses.

In order to achieve that more measurements have to be done to better characterize it. These measurements have to do with the various latencies, the bandwidth as well as the throughput on the overlay. All measurements are done in cycles and are based on the following state machine (fig. 4.1).

Finally, there will be a comparison between the core of the overlay and the implementation of the same kernels on hand drawn designs, implementing them with the same performance floating point units. This comparison will reveal the overhead that is paid in terms of area and frequency, but makes the overlay programmable.

4.2 Test kernels

The kernels that were chosen to be mapped onto the overlay for performance comparisons are all data parallel workloads.

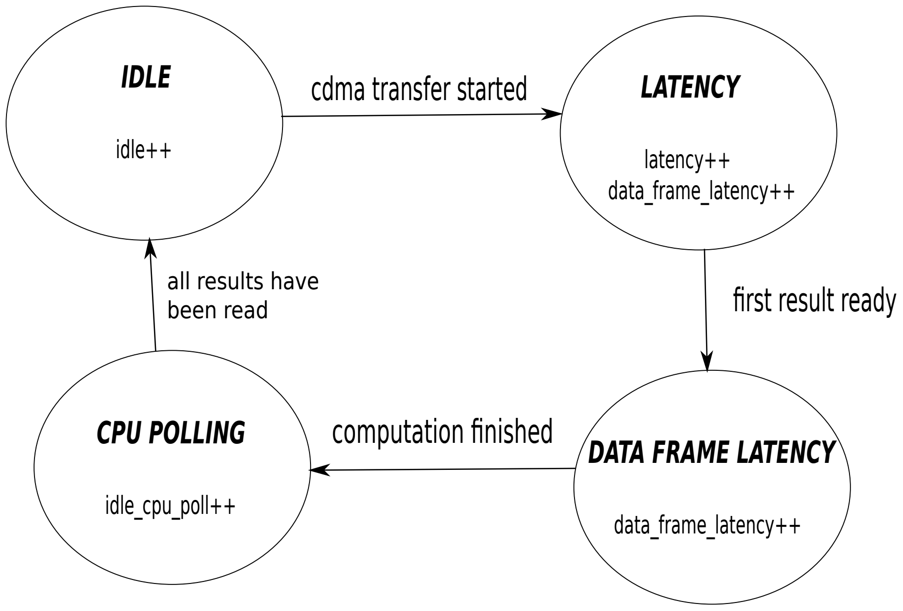


Figure 4.1: State machine that use for testing purposes

The are matrix multiplication, kmeans (sum of absolute differences) [25], 1D Convolution and Stencil operation on matrices. All of them are kernels that are used in scientific workloads.

For the all the kernels it has been able to map one instance to the overlay, except for the stencil that 2 instances were mapped on simultaneously.

4.3 Example of mapping a kernel

Before continuing on showing how to map a kernel for execution on the overlay, it needs to be mentioned that even though a compiler for the project existed it was used for the system that the original authors had build, that was not using the AXI interfaces and the CPU was a OpenSPARC processor. That meant that it would not be immediately used without modifying it, something that was outside of the scope of this thesis. The above means that we had to program each switch and functional unit manually, a process that took a considerable amount of time.

Moving on a particular example will be presented. In the following code section the C code for the convolution algorithm can be seen.

Listing 4.1: Kernel of an algorithm in C

```
for(i=10;i<size;i++){
                    for(j=0;j<10;j++){
                              y[i] += x[i-j] * h[j];
                    }
          }
```

The above code produces the DFG below that is seen in the next figure(fig. 4.2).

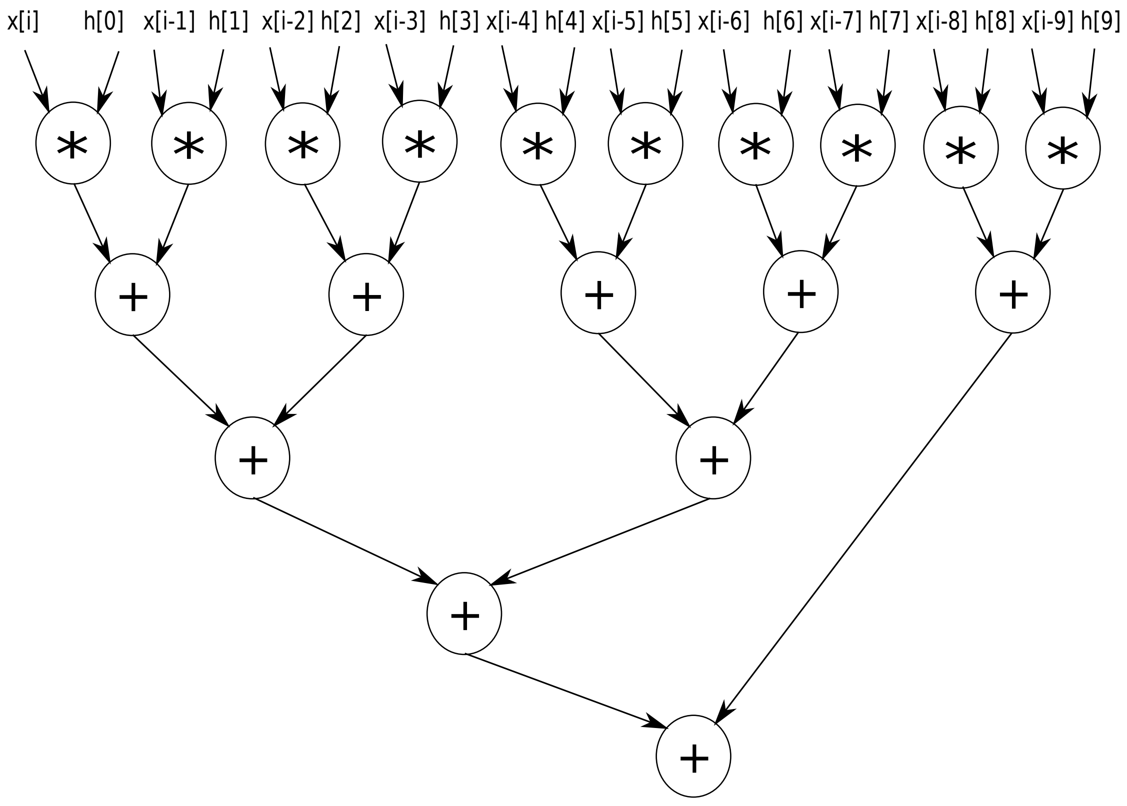


Figure 4.2: DFG produced by the convolution kernel

And finally the schedule on the overlay looks like the one in the following picture(fig. 4.3).

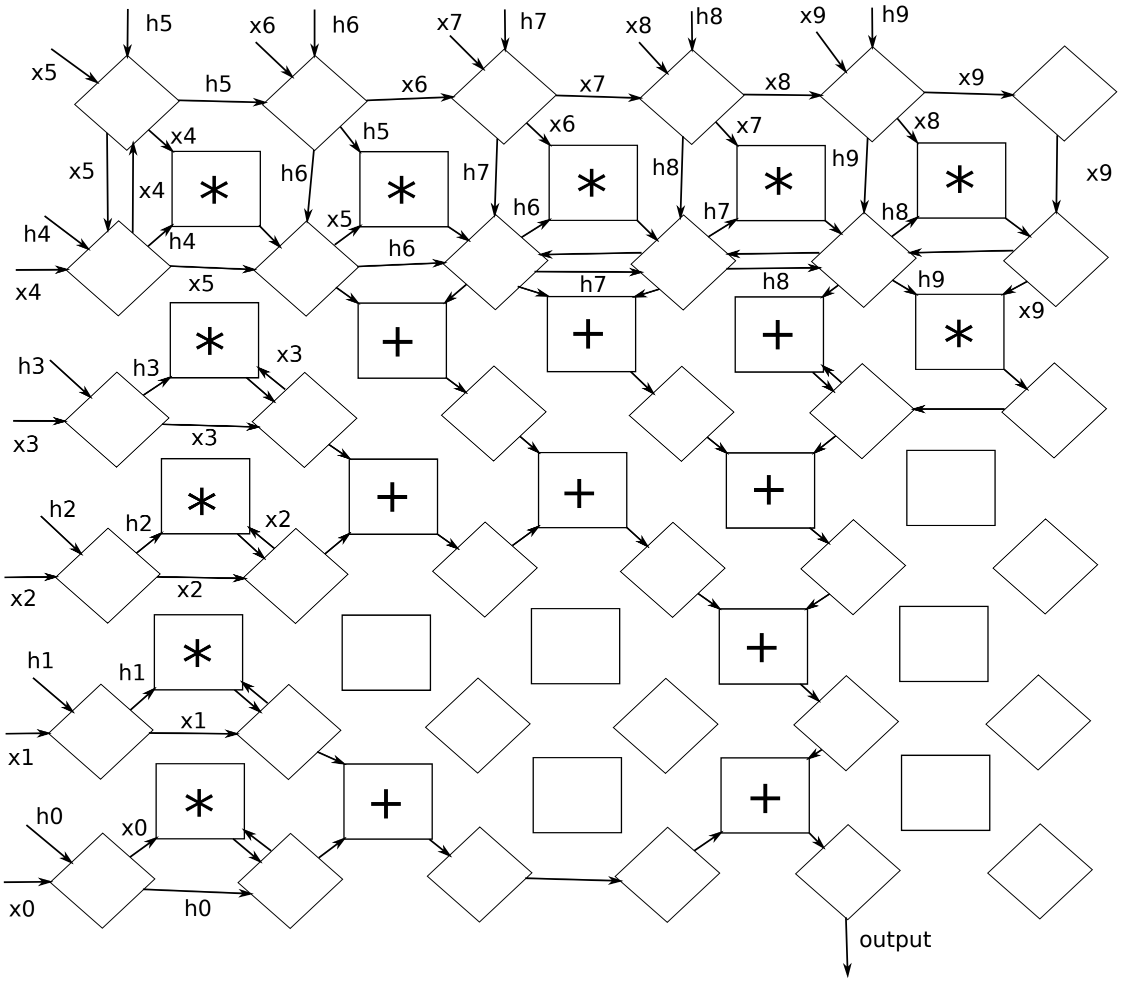


Figure 4.3: Mapping of the Kernel on the Overlay

Results & discussion 5

In this chapter, the results for different kinds of measurements will be presented. The first section will present results related to performance measurements while the second section presents results regarding area measurements and frequency.

5.1 Performance measurements

In this section, 4 versions of the design will be compared

(a) The initial version of the design that operated at 72 Mhz and where one data vector was sent to the overlay for calculation. The AXI bandwidth of this design is 288 Mbytes/s.

(b) A version similar to the first one that had improved pipelining so it could achieve a frequency of 125 Mhz. This gives a bus AXI bandwidth of 500 MBytes/s.

(c) A version with improved frequency but that also makes more efficient use of the bus sending 4 data vectors to the overlay for calculation at each bus transaction. This design has the same bus bandwidth as the previous one.

(d) A version similar to the third one that is using a 64-bit bus instead of a 32-bit. The bus bandwidth of this design is 1000 Mbytes/s.

5.1.1 Latency

In this section, the performance of the overlay in terms of latency (in cycles) is presented in table 5.1. Latency is defined as the number of cycles that are needed for the first result of the overlay to be available. All these measurements were taken using counters inside the peripheral so they are very accurate.

	Design (a)	Design (b)	Design (c)	Design (d)
Matrix mult	62	62	62	49
kmeans	64	64	64	50
Stencil	56	56	56	N/A
Convolution	62	62	62	49

Table 5.1: Latency results for the kernels (in cycles)

As expected, the latency is not affected by the frequency or the amount of data sent per bus transactions; it is only affected by the data width. This is because when a 64-bit bus is used, we can communicate 2 words to the overlay at the same time for computation instead of one at a time in the case of a 32-bit bus. This will allow us to start earlier with the computation of kernels that require multiple inputs.

5.1.2 Throughput

A more interesting metric is the throughput of the overlay. That represents the number of outputs it can process per second. Table 5.2 shows the throughput results for the various kernels, when implementing the kernels on the 4 versions of the design.

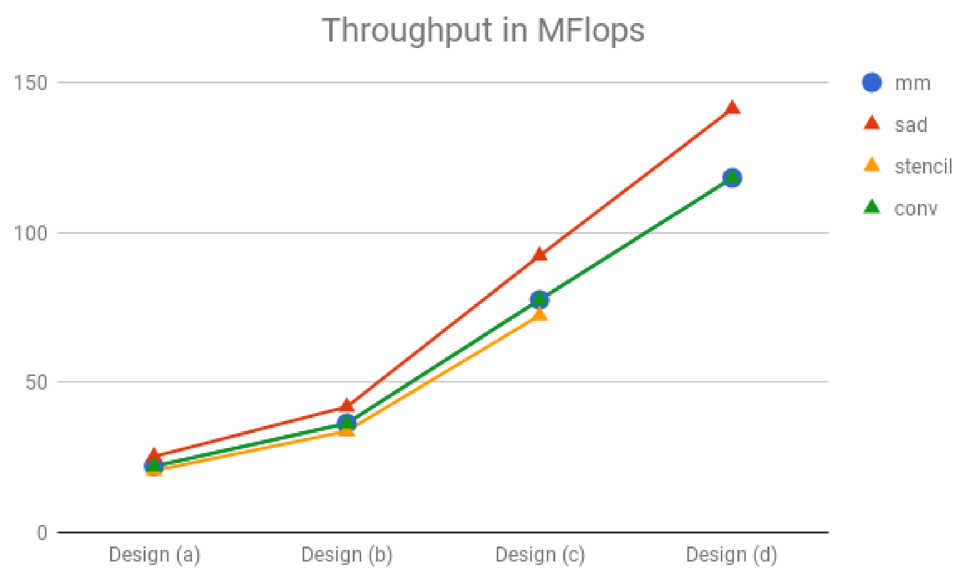


Figure 5.1: Throughput for each kernel implemented on the 4 versions of the design

	Design (a)	Design (b)	Design (c)	Design (d)
mm	22.04	36.33	77.52	118.18
kmeans	25.3	41.86	92.23	141.22
stencil	20.56	33.76	72.32	N/A
conv	22.04	36.33	77.52	118.18

Table 5.2: Throughput for each kernel in MFlops when implemented on the 4 versions of the design

The first observation that can be made from these results, is that throughput scales linearly with the frequency of the design. This is specifically clear when comparing designs (a) and (b), which have the same structure but only differ in their frequency. Designs (a) and (b) process one kernel per bus transactions so they do not take full advantage of the FIFOs interface that is present in the design, causing the the throughput to become tied to the latency.

To overcome this, we send 4 data frames per each bus transaction, thereby fully utilizing the 4-word wide FIFOs that are present in the inputs of the overlay. This way throughput is not tied to the latency of the overlay but the pipeline of the design comes into play and produces results much faster. That can be seen from the comparison between design (b) and design (c) where the same 32-bit bus is used, but on the later design data are readily available to be processed for a longer period of time resulting in a better utilization of the fabric.

Besides the availability of data, another important factor is the bandwidth of the bus that feeds data to the overlay. Design (d) has twice the bandwidth of design (c) resulting in better throughput.

From the available measurements it was observe that the overlay produces a new result (after waiting for the initial latency) at the rates listed in table 5.3.

	32-bit bus	64-bit bus
mm	20	10
kmeans	20	10
stencil	18	9
conv	20	10

Table 5.3: Time between each new kernel result for different bus widths (in cycles)

These numbers together with the frequency of the overlay give us the theoretical maximum performance of the overlay using 32-bit and 64-bit bus widths, in the case that infinite bursts and FIFOs were available. The maximum possible throughput is listed in table 5.5.

	32-bit bus	64-bit bus
mm	118.94	234.08
kmeans	143.98	283.36
stencil	111.52	220.16
conv	118.94	234.08

Table 5.4: Max theoretical performance in MFlops for 32-bit and 64-bit bus

Finally, assuming that we have a bus system that could populate all the 20 ports of the overlay in one cycle, namely a 640-bit wide bus. Then the theoretical maximum performance of the overlay could be achieved, producing a new result each cycle. For a frequency of 125 Mhz, as in the case of design (d), we would be able to achieve an even higher performance as listed in table 5.5.

	Throughput
mm	2.37 GFlops
kmeans	2.87 GFlops
stencil	2.23 GFlops
conv	2.37 GFlops

Table 5.5: Max theoretical performance when using 640-bit wide bus

We also have to mention that in the case of a 640-bit wide bus, the design could also require a slight change in the architecture of the overlay to accept new data every cycle, as this now happens only when a result is ready.

5.2 Speedup

In this section, the speedup of the kernels will be shown when compared with the kernels running on the ARM CPU. The speedup numbers are shown in fig. 5.2, fig. 5.3, fig. 5.4, fig. 5.5.

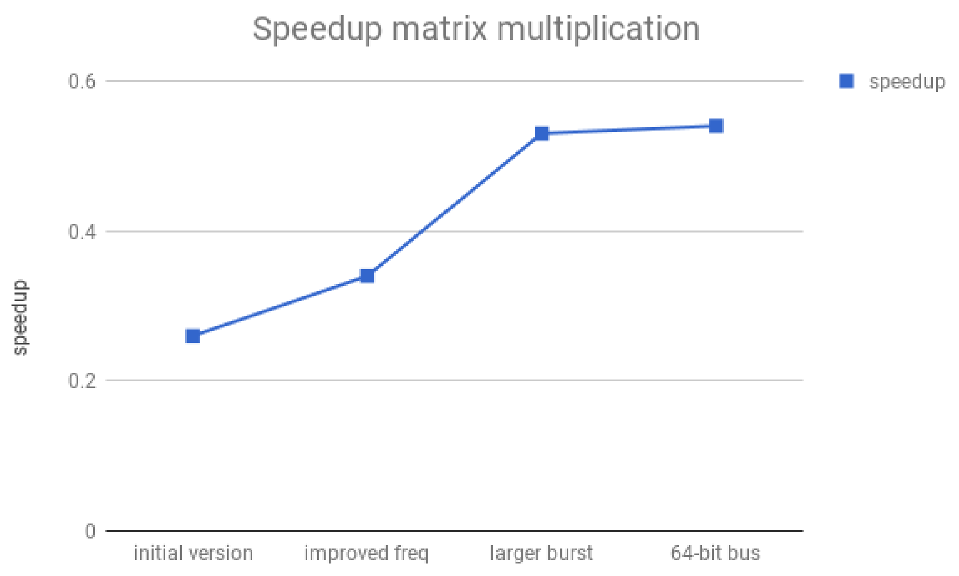


Figure 5.2: Speed up of the matrix multiplication algorithm for different versions of the design

From these results, it can be concluded that the largest jump in performance is observed when bigger bursts are used to transfer data for calculation, which results in

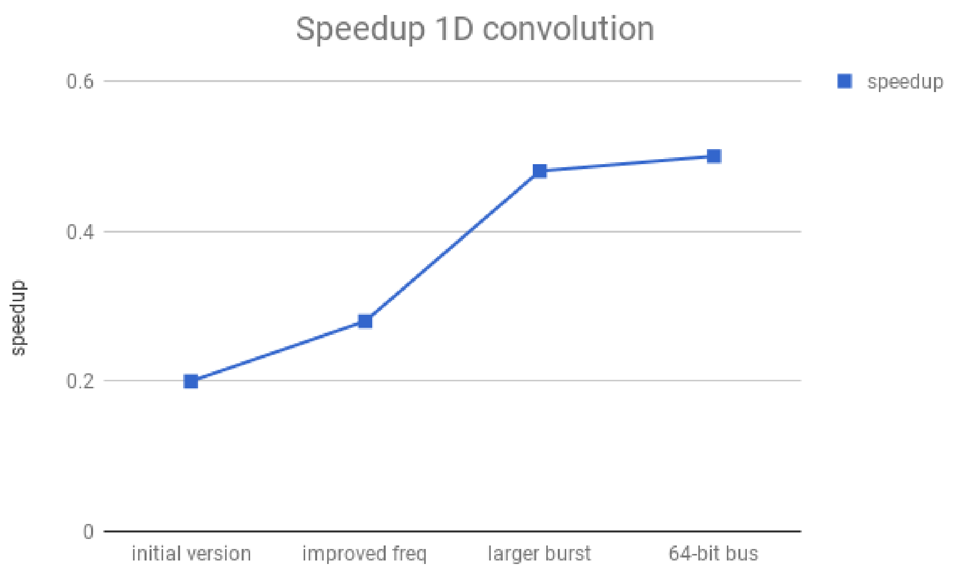


Figure 5.3: Speed up of the 1d-convolution algorithm for different versions of the design

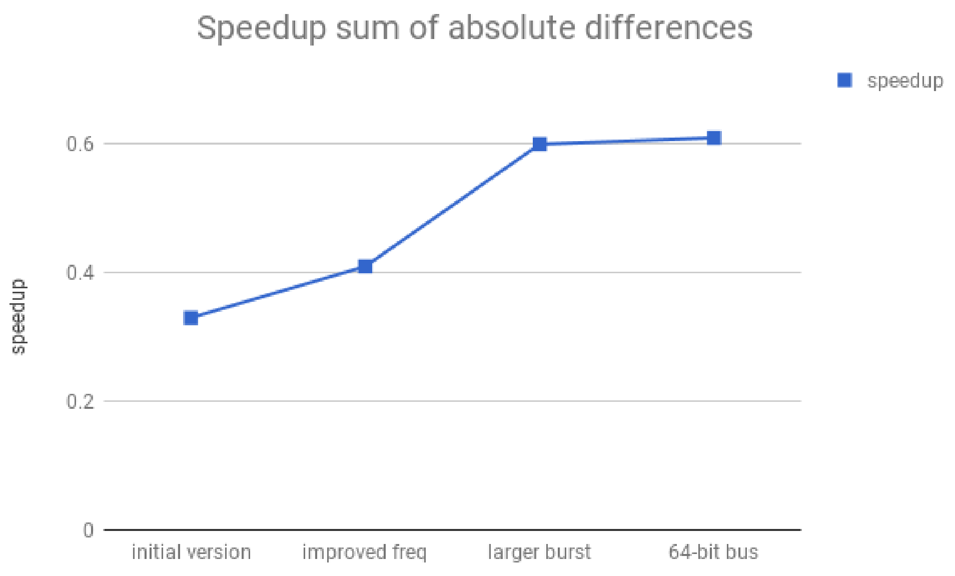


Figure 5.4: Speed up of the kmeans algorithm for different versions of the design

less communication time between the CPU and the peripheral.

The biggest bottleneck was found to be reading data from the main memory. Because data needs to be in a very specific order in memory in order for the overlay to perform calculations, data first needs to be ordered in a way that corresponds to the

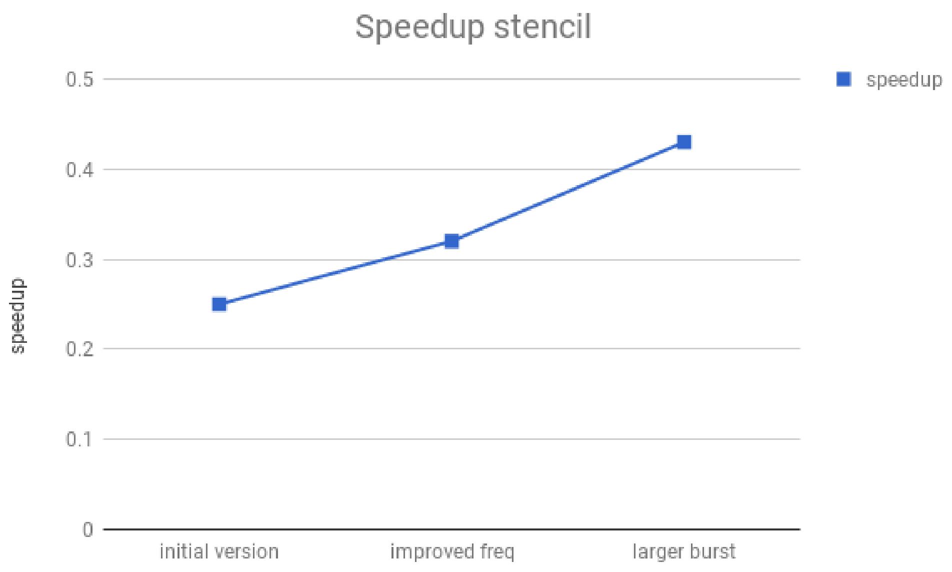


Figure 5.5: Speed up of the stencil algorithm for different versions of the design

physical ports of the overlay.

This intermediate data read and write to a buffer is the biggest bottleneck in performance for this design, and probably is the reason why when reading larger amounts of data gives better performance. This can also be concluded from the profiling of the execution by the time spent in each phase of the execution on fig. 5.6 and fig. 5.7. On top of that, cache needs to be flushed before every memory access as there is no coherency when using the CDMA engine.

A solution to that problem could be to have a local memory build using BRAMs. There, the data to be used would initially be written and rearranged, before they are sent to the overlay for calculation.

In fig. 5.6 and fig. 5.7, "Execution" means that the overlay is receiving data and/or at least one on its functional units is computing a result.

"Idle polling" means that the CPU is reading the results from the FIFOs on the peripheral and writing them to the main memory. During this time, the core of the overlay is not doing any computation.

"Idle memory access" is the time that the CPU needs to read and reorder the data in the main memory so that they are aligned to the physical ports of the overlay to be sent using the CDMA engine (that requires data to be in consecutive memory address) and flushing the cache. This is by far the most time consuming part of the execution

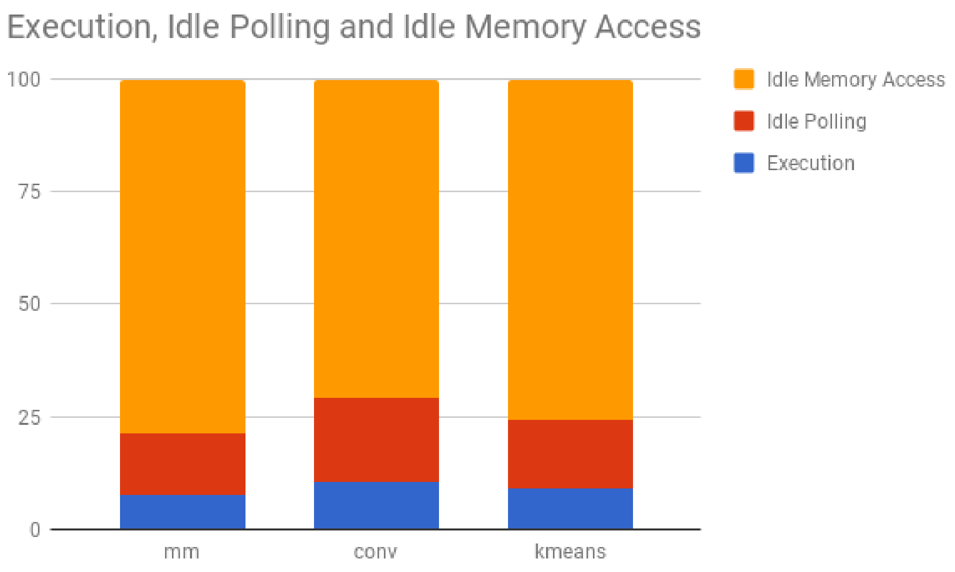


Figure 5.6: Time spent in each of the execution steps for the 64-bit bus overlay

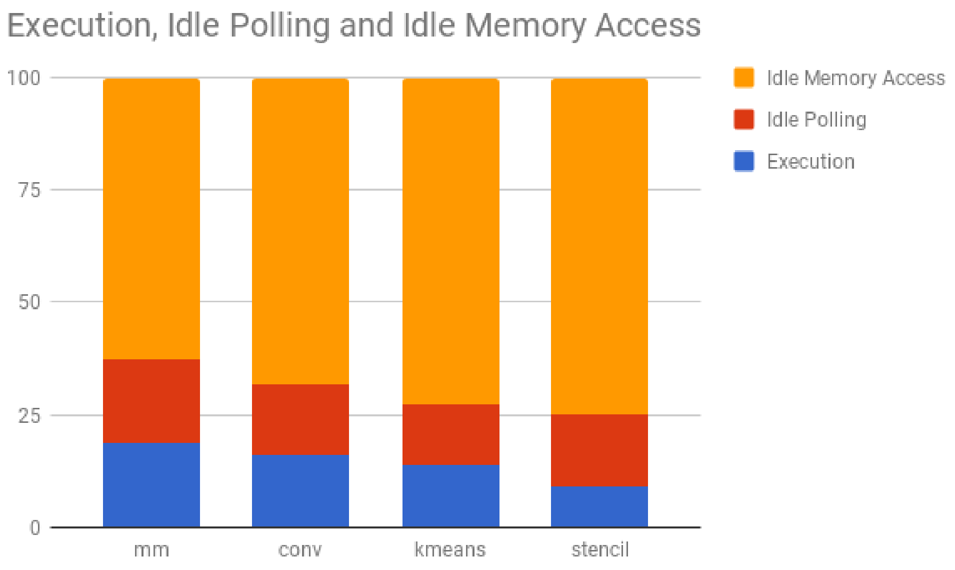


Figure 5.7: Time spent in each of the execution steps for the 32-bit bus overlay

process.

Writing the software in a better way

Based on the results above some of the kernels have been re-written so that they makes more efficient use of the overlay. In the initial version after the CPU sends the command to the CDMA to carry out the transfer, it immediately starts polling the status register for results.

This is not a clever usage, as the results will take some time to be ready. So in the improved version after the command is sent to the CDMA, the CPU starts preparing the data for the next iteration. This way the time spent in idle polling with no reason goes to a meaningful work.

Below the improvements from this reordering can be seen for the 32-bits bus version (Table 5.6). Sadly there was no time for this to be done for all the test cases.

	Old speedup	New Speedup
mm	0.53	0.61
stencil	0.43	0.51
conv	0.48	0.61

Table 5.6: Improved speedup with code reordering

5.3 Reconfiguration

One of the most attractive reasons for using an overlay architecture is the fact that the fabric can be reconfigured in a short period of time in order to support a different function. As mentioned already in a previous chapter, the way the overlay is programmed is by configuration registers forming a large shift register.

When configuration mode is enabled, the configuration registers form a large shift register, such that each cycle shifts 32 new bits of configuration in. So the configuration time is proportional to the size of the overlay and grow linearly to the number of switches and functional units.

Below we can see the reconfiguration time achieved by the overlay. This could have be even lower, but is restricted by the fact that the CDMA engine can only do 16 long burst to a fixed address. Also the configuration time for both the 32-bit and 64-bit buses is the same, because connections internal to the overlay are 32-bit so the 32 extra bits are discarded. Compared with partial reconfiguration of devices that is in the order of tens of milliseconds [5], this is way faster.

	32-bit bus	64-bit bus
Configuration time	11.9us	11.9us

Table 5.7: Configuration time

5.4 Area & Frequency Overhead

In order to quantify the overhead of the overlay in terms of area and frequency, the kernels that were chosen to be mapped on the overlay were also mapped using a handmade implementation.

This was done by generating functional units from the Vivado interface and creating the DFG of the kernel manually, with pipeline registers between the floating point units.

These handmade kernel implementations are going to be compared with the core of the overlay. In essence only the parts that carry out computation are evaluated since both need similar auxiliary logic to communicate with the bus, supporting logic etc.

This comparison will give out the price that is being paid in order for the fabric to be configurable.

	Used	Available	Percentage
LUTs	32499	53200	61.1%
Slice Registers	26414	106400	24.8%
Slices	9138	13300	68.7%
DSPs	100	220	45.4%

Table 5.8: Area usage of the core of the overlay

	Used	Available	Percentage
LUTs	3109	53200	5.8%
Slice Registers	5651	106400	5.3%
Slices	1349	13300	10.1%
DSPs	38	220	17%

Table 5.9: Area utilization of th manual implementation of the matrix multiplication kernel

	Used	Available	Percentage
LUTs	3373	53200	6.3%
Slice Registers	6861	106400	6.4%
Slices	1566	13300	11.8%
DSPs	46	220	20.9%

Table 5.10: Manual implementation of the kmeans kernel

	Used	Available	Percentage
LUTs	2655	53200	5%
Slice Registers	5428	106400	5.1%
Slices	1227	13300	9.2%
DSPs	32	220	14.5%

Table 5.11: Manual implementation of the stencil kernel

Analyzing the results it can be seen that we require 9.6x - 12.22x more LUTs, 3.84x - 4.86x more slice registers and 5.83x - 7.44x more slices.

The overhead depends on the percentage of the fabric that is being used. For example, the kernel for the stencil algorithm uses 16 out out of the 25 functional units therefore it will have a larger overhead than the kernel for kmeans that uses 23 out of them 25 functional units.

The frequency is 68% of the original frequency and the power usage is 2.69x higher on average than a direct implementation of a kernel.

This is a strange result because we see a big difference with the frequency of 125 Mhz that was reported in a previous section. This has to do with the fact that in this case the overlay is synthesized on its own without any supporting logic.

From the manual of the CDMA [22], it can be seen that it can only achieve frequencies up to 120 Mhz on our targeted device. Since the complete design uses the same clock for all the logic, it also forces the overlay to achieve a lower frequency.

6 Conclusion and future work

In this chapter, the work that was performed in this thesis will be summarized as well as suggestions would be provided that can improve the design.

6.1 Conclusions

The goal of this thesis was to have an evaluation of overlay architectures while trying to map such an architecture on a current platform to gain better understanding of what these architectures have to offer and what are their weak points.

Initially, a literature review was conducted in order to find the current status of the topic and look up for alternatives that were suitable for implementation. Based on this literature review, a design was identified that met the goals of the thesis. Then, a study was done in order to understand how the design can be programmed to execute our targeted benchmarks. The design was subsequently modified in order to make it fit within the available FPGA as well as extend its functionally in order to support floating point operations using the Xilinx DSP blocks. The switching network also was slightly modified and saw a drop of 20% in area usage.

When finished with the modifications of the design and its correctness was evaluated, it had to be connected to the ARM CPU via the various AXI interfaces available on the SoC. The design was treated as a peripheral to the CPU which can program it and send data to it in order to to offload calculations.

A number of kernels were then adapted to this programming scheme to evaluate the performance but also the programmability of the systems. Although we saw no performance gains when compared to the direct CPU implementation, we characterized the design based on a number of different measurements and pointed the weak points of the current integration. Based on these results and with a small number of optimizations that were proposed, a big performance gain can be observed.

Finally, we confirmed the advantages of the overlay architectures when it comes to reconfiguration time and also pointed out the trade-offs they come with.

Our results show that our architecture can be reconfigured in only 11.9us, as compared to seconds for full FPGA reconfiguration. However, the overlay architecture uses 10.5x more LUTs and causes a drop in frequency of about 30% for the chosen architecture.

Our initial goal of having an initial mapping and evaluation of an overlay architecture was achieved, while there is still room for improvement.

6.2 Future work

- The current interconnection scheme is the biggest bottleneck of the architecture. It is too general and for most kernels when it is not needed, as most of the connections are not used especially in the later stages of the DFG, as can be seen in fig. 4.3.

  Adoption of the interconnection scheme presented in [13] could improve the frequency as well as the area usage, but still there will be a 4-5 times higher LUT usage than the manual implementation.

- At every switch on the left and top side only two inputs are available. Initially, when using a 4x4 size to map kernels, the inputs were found to be enough. But as the overlay got bigger the number of inputs was found not to be sufficient, because they are 2*N while the functional units are N*N. As N gets bigger there will be a bigger gap between those two numbers.

  Extending the input to also cover the right hand side could help in solving that problem and would also help the functional units on the right hand side as they are currently hard to access.

- This version of the overlay had 4-word wide FIFOs per input and output. Using bigger FIFOs could improve the usage and result in lower communication bottlenecks with the main memory.

- Using a local memory on the FPGA side will lead to large performance gains, as currently the utilization of the design is low. As seen by the figs. 5.6 and 5.7 only 20% and 10% of the time is spent on execution for the 32-bit and 64-bit cases respectively.

  If the fabric was fully saturated we could have had speedups of about 3x for the 32-bit case and 6x for the 64-bit case.

- Two thirds of the core are currently occupied by the switch and only one third by the 2 floating point units. This means that the functional units can be extended to include one more floating point unit, even potentially two.

  The floating point unit that could be the most useful for this purpose, is a floating point comparator so we can have comparisons on the fabric itself without the need of the CPU.

- Currently there is a bug in the 64-bit version of the design and transactions on the bus sometimes are not carried out. Probably this has to do with the way Xilinx drivers handle sending 64-bit bursts with the CDMA, combined with the way that the bare metal application handles the malloc system call that was used to allocate memory.

On the 32-bit version, 0.1% of results are different between runs with the same data and this probably is caused by how the cache is flushed. It has to be pointed out though that both of these bugs do not in any way interfere with the performance measurements that were conducted.

Also there is a bug with the overlay itself, that when two or more numbers enter the overlay and using functional units with more than a 4-cycle latency, the flow-control system stops propagating data through the pipeline.

Bibliography

[1] Z. Marsec, "Detailed performance evaluation of data-parallel workloads on the dyser prototype system," in *M.Sc. Thesis*.

[2] L. H. Crockett, R. A. Elliot, M. A.Enderwitz, and R. W.Stewart, "The zynq book," August 2015.

[3] "Zynq 7000 all programmable soc technical reference manual, ug585 (v1.11) september 27, 2016."

[4] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *CODES+ISSS'10*, Stottsdale, Arizona, October 2010.

[5] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant fpga compilation," in *IEEE Embedded Systems Letters Vol.3, No.3*, September 2011.

[6] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from opencl using reconfiguration contexts," in *IEEE Micro*, January/February 2014.

[7] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmabality," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[8] J. Peltenburg, A. Hesam, and Z. Al-Ars, "Pushing big data into accelerators: Can the jvm saturate our hardware?" in *Proc. International Workshop on OpenPOWER for HPC*, Frankfurt, Germany, November 2017, pp. 220–236.

[9] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the bwa-mem dna alignment algorithm," in *Proc. International Conference On Computer Aided Design*, Austin, USA, November 2015, pp. 240–246.

[10] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the pairhmm forward algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Shenzhen, China, December 2016.

[11] J. Hoozemans, R. Heij, J. van Straten, and Z. Al-Ars, "Vliw-based fpga computation fabric with streaming memory hierarchy for medical imaging applications," in *13th International Symposium on Applied Reconfigurable Computing (ARC2017)*, Delft, The Netherlands, April 2017.

[12] V. Govindaraju, C.-H. Ko, and K. Sankaralingam, "Dynamucally specialized datapaths for energy efficient computing," in *IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[13] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architectures based on dsp blocks," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[14] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughtput oriented fpga overlays using dsp blocks," in *2016 Design, Automation and Test in Europe Conference and Exhibition(DATE)*, 2016.

[15] A. K. J. et al, "Deco: A dsp block based fpga accelerator overlay with low overhead interconnect," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2016.

[16] A. K. Jain and al, "Adapting the dyser architecture with dsp blocks as an overlay for the xilinx zynq," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies(HEART2015)*, Boston, MA, June 2015.

[17] "Dyser release v1.0." [Online]. Available: http://research.cs.wisc.edu/vertical/dyser-release-v1/doku.php

[18] V. Govindarajn and al, "Dyser: Unifying functionallity and parallelism specialization for energy-efficient computing," in *IEEE MICRO*, Sept - Oct 2012.

[19] "Zynq-7000 all programmable soc." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[20] "Axi reference guide, ug761 (v13.1) march 7, 2011." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[21] "Floating-point operator v7.1." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf

[22] "Axi central direct memory access v4.1." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf

[23] "Virtex ultrascale+." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable

[24] "Impact benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil/parboil.aspx

[25] "C source code implementing k-means clustering algorithm." [Online]. Available: https://www.medphysics.wisc.edu/~ethan/kmeans