Streaming FPGA Based Multiprocessor Architecture for Low Latency Medical Image Processing

Roelof Willem Heij

CE-MS-2016-15

Abstract

In this work a fast and efficient implementation of a Field Programmable Gate Array (FPGA) based, fixed hardware, streaming multiprocessor architecture for low latency medical image processing is introduced. The design of this computation fabric is based on the ρ-VEX Very Long Instruction Word (VLIW) softcore processor and is influenced by architectures of modern Graphics Processing Unit (GPU) implementations. The computation fabric is capable of exploiting several types of parallelism, including pipelining, Instruction-level Parallelism (ILP) and Data-level Parallelism (DLP). The multiprocessor in the fabric is implemented by a chain of ρ-VEX processors that function as a processor pipeline. A memory architecture to support the high throughput of this processor pipeline has been created, making the computation fabric capable of stream processing. The basic building blocks of this memory architecture are single cycle accessible, dual port scratchpad memories. A total of 16 instances of the computation fabric are implemented on a Virtex-7 FPGA, creating an array of multiprocessors that is capable of processing 43.52 images per second when running a typical medical image processing algorithm workload on an operating frequency of 193 MHz. This makes the implementation suitable for real-time medical image processing. The processor pipeline depth of the computation fabric is generic, and can be changed according to the requirements posed by the algorithm workload. This makes the architecture flexible and general enough to handle changes and updates to the algorithm workload.

TU Delft Delft University of Technology

CE Lab Computer Engineering Laboratory

Streaming FPGA Based Multiprocessor Architecture for Low Latency Medical Image Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Roelof Willem Heij
born in Krimpen aan den IJssel, The Netherlands

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

Streaming FPGA Based Multiprocessor Architecture for Low Latency Medical Image Processing

by Roelof Willem Heij

Abstract

In this work a fast and efficient implementation of a FPGA based, fixed hardware, streaming multiprocessor architecture for low latency medical image processing is introduced. The design of this computation fabric is based on the ρ-VEX VLIW softcore processor and is influenced by architectures of modern GPU implementations. The computation fabric is capable of exploiting several types of parallelism, including pipelining, ILP and DLP. The multiprocessor in the fabric is implemented by a chain of ρ-VEX processors that function as a processor pipeline. A memory architecture to support the high throughput of this processor pipeline has been created, making the computation fabric capable of stream processing. The basic building blocks of this memory architecture are single cycle accessible, dual port scratchpad memories. A total of 16 instances of the computation fabric are implemented on a Virtex-7 FPGA, creating an array of multiprocessors that is capable of processing 43.52 images per second when running a typical medical image processing algorithm workload on an operating frequency of 193 MHz. This makes the implementation suitable for real-time medical image processing. The processor pipeline depth of the computation fabric is generic, and can be changed according to the requirements posed by the algorithm workload. This makes the architecture flexible and general enough to handle changes and updates to the algorithm workload.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-15

Committee Members :

Advisor:	dr. ir. Zaid Al-Ars, CE, TU Delft
Chairperson:	dr. ir. Stephan Wong, CE, TU Delft
Member:	dr. ir. Chris Verhoeven, ELCA, TU Delft

Dedicated to my fiancée Sarah, and to my parents

Contents

List of Figures

List of Tables

List of Acronyms

AHB Advanced High-performance Bus

ALU Arithmetic Logic Unit

AMBA Advanced Microcontroller Bus Architecture

ASIC Applications Specific Integrated Circuit

BRAM Block Random Access Memory

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DLP Data-level Parallelism

DSP Digital Signal Processing

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

ILP Instruction-level Parallelism

ISA Instruction Set Architecture

LUT Look-Up Table

MIG Memory Interface Generator

OpenCL Open Computing Language

NOP No Operation

PCIe Peripheral Component Interconnect Express

PTX Parallel Thread Execution

RAM Random Access Memory

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

SP Streaming Processor

SRAM Static Random Access Memory

TLP Thread-level Parallelism

TPC Texture/Processor Cluster

VEX VLIW Example

VHDL Very High Speed Intergrated Circuit Hardware Description Language

VLIW Very Long Instruction Word

Acknowledgements

There are a lot of people that have helped me in the course of my thesis project. I want to express my gratitude to all of you.

First I would like to thank my thesis advisor Zaid, for his endless patience and support. Even at times when the work on my thesis project had seemingly come to a standstill and I was completely drained of motivation, he would be nothing but enthusiastic and encouraging. I would also like to thank Stephan Wong for chairing my committee, and Chris Verhoeven for being part of my committee.

Joost and Jeroen have also been extremely helpful and supportive. Whenever I encountered trouble or issues in my work, I could always count on them to help me sort them out.

I would also like to thank all the marvelous master students that have accompanied me in the MSc Lab of the CE group. Especially Haji and Koray have really helped and motivated me in the final months of my thesis project.

A special thanks goes to my parents, for supporting me in every possible way, in spite of the enormous burden they bear.

Finally I would like to express my gratitude to my wonderful fiancée Sarah. She has shown an incredible amount of patience and support on a daily basis. Words can't describe what she means to me.

Roelof Willem Heij
Delft, The Netherlands
November 27, 2016

1 Introduction

1.1 Context

Image processing has become increasingly complex and resource consuming over the years. Traditionally, the computers Central Processing Unit (CPU) took care of all operations in a computer system, including image processing. In 1999, the introduction of a designated piece of hardware suited for image processing on desktop computers, mainly referred to as a Graphics Processing Unit (GPU), marked a new era for image processing [2]. Nowadays there are various forms of processors which are widely used in everyday life. These forms include CPU, GPU, Applications Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA). There is no single best solution for every image processing task, so for each application one should consider all options in order to acquire optimal performance.

Besides increasingly complex and resource consuming, image processing is also becoming more common and crucial in many sectors. Examples include health care, security and computer vision. Because of the high social, industrial and academic value of image processing, a tremendous amount of research aiming to make image processing faster and more efficient is conducted. An example of this is the Almarvi (**Al**gorithms, Design Methods, and **Ma**ny-core Execution Platform for Low-Power Massive Data-**R**ate **V**ideo and **I**mage Processing) project. This project is a collaboration between several technical universities and several leading companies, and aims to reduce some of the technical difficulties in this area [3]. The Delft University also takes part in this project. To some extend, the work for this thesis project is conducted as a part of the Almarvi project.

1.2 Problem definition

Image processing tasks are considered to be relatively compute intensive, and can consume vast amounts of resources and time when performed in traditional, sequential ways of computation. This is a problem in certain cases, for instance whenever images have to be processed in real time. An example case where real time processing is paramount is in medical imaging. Medical imaging is the technique that aims to visualize the interior of a body. This technique has numerous clinical advantages for both the patient and the surgeon. However, the high computational complexity of medical imaging makes real time processing in a traditional way impossible.

Luckily some medical imaging processing algorithms are suited for parallel execution. By parallelizing the execution of these algorithms, their execution time can be drastically reduced. Using hardware that can provides good conditions for parallel execution, real time processing can be made possible. The goal of this thesis project is to design a

computation fabric that is able to make real time image processing for medical imaging applications possible.

One of the requirements for medical image processing hardware (or so-called computation fabrics[1]) stated by Philips in the Almarvi project is that the performance of the computation fabric desired for their commercial medical imaging product has to be ensured over time. Medical imagers have a high expected lifetime of up to twenty years, while that of the computation fabrics that support these products has a lifetime of around five years. This unusually long lifetime requirement poses challenges on the development and maintenance cycle of these computation fabrics. Since performance portability between computing platforms[2] cannot always be ensured and re-designing, re-optimizing and re-testing of computing platforms, it is desirable to have a fixed hardware platform.

Another requirement is that the medical image processing hardware should be able to process images real-time. For this to be possible, a minimum performance of 15 processed frames per second should be achieved. To allow the human eye to optimally interpret the stream of processed images, a performance of 30 frames per second is desired.

Based on this problem definition, the following research questions can be formulated.

- Is it possible to design and implement a fixed hardware image processing computation fabric that meets the requirements to be used in commercial grade medical imaging products?

- Can we create a computation fabric that is general enough to handle changes and updates to the image processing algorithms used in the field?

1.3 Design constraints

The starting point of the computation fabric is the ρ-VEX Very Long Instruction Word (VLIW) softcore processor that is being developed by the Computer Engineering Department of the Delft University of Technology [4] [5]. Choosing the ρ-VEX processor as a starting point for our fabric mainly serves academic purposes. However, a recent study indicated the remarkable processing power of the ρ-VEX processor in comparison to other, commercial grade softcore processors [6]. For academic purposes the source code of this processor is made available, enabling us to implement it in our image processing computation fabric [7].

The computation fabric should preferably be implemented on an FPGA. There are two reasons for this. Firstly, Philips has indicated to be interested in FPGA-based solutions. Secondly the ρ-VEX processor developed by the Delft University of Technology is a softcore processor that is designed and maintained on an FPGA. Because of this, a complete toolchain and development environment are already available and ready for use on an FPGA.

[1] We define a computation fabric as a piece of hardware dedicated to a certain task, like processors and accelerators.

[2] We define a computing platforms as the physical types of hardware a computing fabric can be implemented on. Examples are ASIC and FPGA.

1.4 Related work

Acceleration of image processing tasks is a large field in computing. A prior study on using the ρ-VEX processor for image processing applications is performed in [6]. This thesis project aims to fulfill parts of the references for future work mentioned in that paper.

There are numerous works on FPGA based image processing. In [8], an FPGA-based compute fabric is proposed using the LE-1 softcore, which is based on the same Instruction Set Architecture (ISA) as the ρ-VEX, targeting medical image processing applications. This work focuses solely on offering a highly multi-threaded platform without providing a memory hierarchy that can sustain the needed bandwidth through the pipeline. A related study on accelerating workloads without compromising programmability is [9], with one of the design points being a convolution engine as processing element. A well-known prior effort, and one of the inspirations of this work, uses softcore processors to provide adequate acceleration while staying targetable by a high level compiler is the Catapult project [10]. The target domain is ranking documents for the Bing search engine. A related effort that aims to accelerate Convolutional Neural Networks is [11]. However, that project did not aim to conserve programmability but only run-time reconfigurability, as the structure of this application does not change enough to require this.

We are not aware of any other implementation of an independent FPGA based computation fabric that supports streaming medical image processing. None of the related works shows an implementation or a design of an independent FPGA based image processor that uses streaming. Existing implementations and designs related to streaming image processing on FPGAs all function as accelerators or co-processors [12][13][14], while existing implementations and designs of independent (image) processors on FPGAs do not use streaming [15][16][17][18][19].

1.5 Thesis outline

Chapter 2 provides background information on the characteristics of the ρ-VEX softcore processor and on image processing accelerators. A capability analysis based on this information will be used to point out what characteristics the desired computation fabric should implement. Chapter 3 first gives a short introduction to algorithm analysis of both sequential and parallel algorithms. This is followed by a broad overview of image processing algorithms. After this, an algorithm workload for medical imaging purposes is constructed. Chapter 4 describes the design of the computation fabric based on the requirements posed by the constructed algorithm workload that is formulated in the previous chapter. Chapter 5 shows how the design is implemented on an FPGA that is able to communicate with several peripherals and a host machine. Chapter 6 describes the measurement setup for the computation fabric and reports the results of these measurements. Chapter 7 ends this thesis with a conclusion on the achieved results and recommendations for future work.

2 Processors for image processing

The aim of this thesis project is to design and implement a computation fabric that is capable of fast and efficient execution of image processing algorithms. Preferably it should also meet the desired requirements for a real-life use case, the Interventional X-Ray System by Philips. Preferably the computation fabric should also be suited for executing general purpose tasks. In order to be able to describe a design for this fabric, it might be helpful to look into the designs of other (image) processors. Also looking into other computation platforms might be helpful, even though the computation platform for our computation fabric has already been chosen to be a Field Programmable Gate Array (FPGA). Investigating other computation platforms will help to identify the strong points and weak points of the FPGA compared to the other computation platforms, which will in turn help us with design considerations when creating the computation fabric.

This chapter will start with a computation platform comparison, exploring the pros and cons of four computation platforms, namely the Central Processing Unit (CPU), Graphics Processing Unit (GPU), FPGA and Applications Specific Integrated Circuit (ASIC). This is followed by a study on two processors relevant to our computation fabric. These are the ρ-VEX processor, that will be used as starting point for our fabric, and the GPU, which is designed with a special focus on image processing. The chapter is concluded by a discussion on the pros and cons of the ρ-VEX and the GPU in light of our target application, in order to determine what parts and properties are desirable for implementation in our fabric. [36–46]

2.1 Computation platform comparison

Both Philips and the TU Delft have indicated that the FPGA is the platform of their preference. However, as stated in Section 1.2, processors and accelerators can be implemented on various types of hardware. There is no single best platform for all applications types, since all of them have very different capabilities and limitations. For this reason, all of the possible platforms will be shortly examined and compared in terms of various characteristics. They are summarized below.

- **Performance** This metrics indicates the capability of processing operations and data in a certain amount of time. Higher is better.

- **Performance portability** This metric indicates how well performance is maintained whenever the code that needs to be executed is ran on another device of the same type. Higher is better.

- **Scalability** This metric indicates the capability of improving the overall performance by adding more hardware of the same sort. Higher is better.
- **Applicability** This metric indicates how wide the range of applications is that can be executed efficiently. Higher is better.
- **Development time** This metric indicates the time needed for developing an implementing of a processor or accelerator and/or the porting of needed software to this processor or accelerator. Less time is obviously better. However, to avoid confusion in the overview; higher is better.
- **Cost** The metric indicates the cost of the hardware itself and does not take development time or other costly factors into account. Less costs are obviously better. However, to avoid confusion in the overview; higher is better.

CPU This platform is probably the most well-known and widespread of all. The ubiquitous CPU is present in every computer, where it functions as the main controller of the system. In order to fulfill this role, the CPU is capable of executing an enormous amount of different operations and tasks. It is able to handle and perform tasks with high complexity in an efficient way. However, the CPU is not able to process vast amounts of data or instructions per second because this is fundamentally in conflict with the support of high complexity tasks. Performance is scalable to a certain extent. CPUs of the same brand are in most cases backwards compatible and will deliver equal or better performance when an application that is written for a certain CPU is ported to a newer CPU. The development time for CPUs is low. They can be targeted using a large range of programming languages. The cost of purchasing a consumer grade CPU is relatively low.

GPU The time that GPUs were used for image processing only is long gone. The capability of GPUs to be used as accelerators for all sorts of algorithms and other compute intensive tasks is being exploited widely, especially in academia and the industry. A small range of large problems can be executed orders of magnitude faster on GPUs because of their ability to process huge amounts of data at once. However, portability is a big issue. Equal or better performance can not be guaranteed when porting a program to a newer GPU. In fact, there have even been reports of reduced performance when porting programming code to newer devices. The performance of GPUs is very scalable. The existence of GPUs of over a thousand cores and systems with multiple GPUs support this fact. Development time for GPU is relatively low thanks to the programming languages that allow programmers to target GPUs in an easy and familiar way. The cost is comparable to that of the CPU.

FPGA While CPUs and GPUs are fixed pieces of hardware, FPGAs are reprogrammable. Users can design their own desired hardware and upload it to the FPGA, and revise it if necessary. This allows users to create piece of hardware that are tailored to their needs. The FPGA is a large array that contains three types of basic building blocks. These are the flip-flop register (mostly referred to as *Register* or *Reg*),

the Look-Up Table (LUT) and the Block Random Access Memory (BRAM). A synthesis, implementation and routing tool can translate a hardware design to digital logic that can be implemented in this array. The designs can be created in hardware programming languages like Very High Speed Intergrated Circuit Hardware Description Language (VHDL) or Verilog, which have stood the test of time very well. This guarantees the portability of hardware designs on FPGAs. Due to their nature FPGAs do not support high clock frequencies like the other platforms, but compensate for this by being optimized for the specific task that they are designed for. Research shows that acceleration on FPGAs is scalable. Unfortunately development time of custom hardware designs is extremely high, taking up to multiple years for complex designs. The price of FPGA development boards is slightly higher than that of CPUs or GPUs, but is still relatively low.

ASIC When implementing a custom VHDL or Verilog hardware design, ASIC is also an option. ASIC are generally faster than FPGAs because they can run at a higher frequency, but are not reconfigurable and generally have an even longer development time. The production costs of producing a single ASIC are extremely high, but become lower when producing larger quantities. ASICs theoretically have the same level of performance portability as FPGAs, but typically outperform FPGAs in terms of scalability.

Conclusion Figure 2.1 and Table 2.1 show that FPGA is indeed a viable option given that high performance portability, applicability and scalability are must-haves. For this reason the FPGA based ρ-VEX processor will be used as a starting point for the design of the accelerator. However, since the GPU is designed especially to support the processing of data in parallel by incorporating massive multithreading, inspecting this platform might provide viable information on architectural design choices when designing the image processing accelerator.

Table 2.1: Overview of computation platform exploration

Platform	**CPU**	**GPU**	**FPGA**	**ASIC**
Performance	+	++	-	+/-
Performance portability	+	-	++	++
Scalability	-	++	+	++
Applicability	+/-	-	++	+
Development time	++	+	-	–
Cost	+	+	+	-

2.2 ρ-VEX analysis

For academic purposes it is desired that the main building block for our fabric is the ρ-VEX softcore Very Long Instruction Word (VLIW) processor developed by the Computer Engineering department of the Delft University of technology that is based on the VLIW Example (VEX) Instruction Set Architecture (ISA). In order to design an architecture that incorporates this processor, it is important to understand its working, capabilities

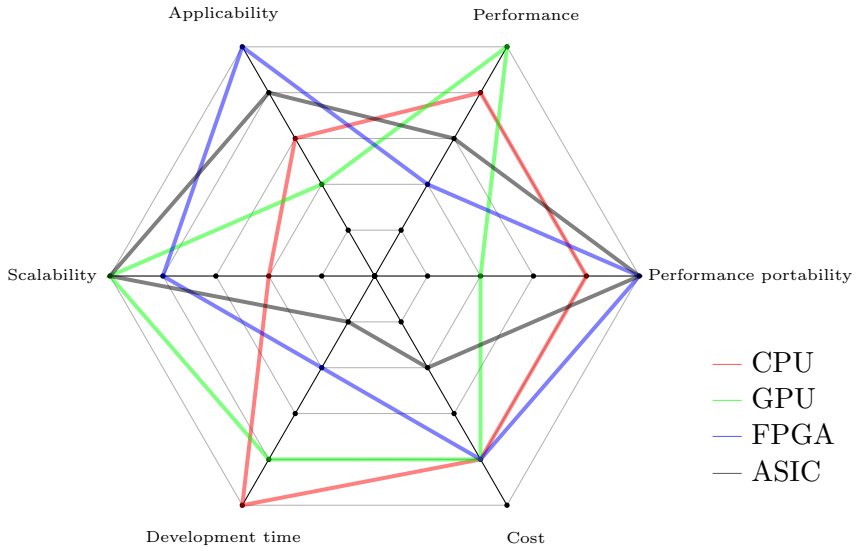


Figure 2.1: Graphical overview of computation platform exploration for

and limitations. This section will first explain the concepts of VLIW. This is followed by a description of the architecture and working of the ρ-VEX processor.

2.2.1 VLIW design philosophy

VLIW is not a precisely defined methodology, but rather an architectural design philosophy. The VLIW methodology was introduced by Joseph Fisher et al. [20]. VLIW processors are naturally capable of exploiting Instruction-level Parallelism (ILP)[1], due to the fact that they are able to issue multiple instructions at the same time. In this way a VLIW processor is able to allow higher performance without adding any complexity to its design while doing so. Exploiting this form of parallelism does require an ILP-oriented compiler that can issue certain combinations of instructions in parallel, and possibly out of order.

Because of its natural support of parallelism, VLIW processors are promising for image processing tasks. One of the parameters that dictate the VLIW's capability of handling parallelism is its issue width. This indicates the width of the input size of the processor, and dictates how much operations can be processed in parallel. Note that the VLIW processor can only execute operations in parallel if the VLIW processor has enough resources available to perform all of them, and if there are no data dependencies between the operations.

2.2.2 ρ-VEX design

Along with the VLIW design methodology, Joseph Fisher also introduced an accompanying ISA, called VEX. The Computer Engineering group of the Delft University of

[1]ILP aims to increase the speed of a computing fabric by issuing and executing multiple instructions in parallel.

Technology implemented a reconfigurable VLIW processor based on the VEX ISA, and called the ρ-VEX processor. This section will discuss the architecture and characteristics of the ρ-VEX. The current design of the ρ-VEX is intended to be implemented on an FPGA, which is connected to a regular x86 host machine through a USB connection. Note that, even though the FPGA that implements the ρ-VEX processor is connected to a host machine, the ρ-VEX is an independent processor. The host machine is only used to program the FPGA and hand the scheduled instructions to the ρ-VEX.

2.2.2.1 Reconfigurability

The spear points of the ρ-VEX processor are the capabilities of reconfigurability and parameterizability. These allow the ρ-VEX processor to be tailored for a specific application in order to achieve high performance without having to redesign the processor in any way. The ρ-VEX is can be dynamically reconfigured and parameterized at design-time, and in same cases even at run-time. In its full form, the ρ-VEX is an eight-issue VLIW processor, created by combining four two-issue cores. Using this structure, the issue-width of the ρ-VEX can be parameterized to either two, four or eight. Moreover, the ρ-VEX supports an number of configurations that emulate multi-core behavior. When configured as a multicore processor, all cores in the ρ-VEX work completely individual and have their own caches and memories. Possible configurations for the ρ-VEX are listed below.

The ρ-VEX can be configured as:

- A single eight-issue processor
- A single four-issue processor
- Two four-issue processors
- A single two-issue processor
- Two two-issue processors
- Four two-issue processors
- One four issue-processor and two two-issue processors

2.2.2.2 Processor architecture

The architecture of the ρ-VEX processor cores is an adaptation of the VEX processor, an example implementation of a VLIW processor. It is an ST200 series core based on technology jointly developed by Hewlett-Packard Laboraties and STMicroelectronics [21]. The VEX processor was created by the Joseph Fisher, the inventor of VLIW philosophy and employee of Hewlett-Packard Laboratories, to demonstrate the working of his VLIW ISA, which is simply called the VEX ISA. This ISA is also used by the ρ-VEX processor. Also the toolchain that was developed and released to accompany the VEX processor is being used as a part of the ρ-VEX toolchain [22].

In its full form the ρ-VEX processor features four two-issue ρ-VEX cores working together as a single eight-issue processor. Each of the two-issue cores implements a seven-stage pipeline that supports instruction forwarding. Each core has its instruction and data memory, implemented in the form of either caches of single-port memories. The following resources are available for implementation in each core:

- Arithmetic Logic Unit (ALU)
- Multiplication unit
- Branch unit
- Load/Store unit
- General Register file (GR) consisting of 64 32-bit registers with 4 write and 8 read ports
- Branch Register file (BR) with 8 1-bit registers

Out of the first four resources, the core can support a maximum of two. The last two are present in every core.

2.2.2.3 Communication and peripherals

The ρ-VEX processor allows internal communication between the cores, and between the ρ-VEX processor and peripherals. Whenever the processor is operating with an issue-width of four or eight, the collaborating cores can communicate over an all-to-all crossbar, relieving stress on the Advanced High-performance Bus (AHB) bus. Each core is connected to an AHB bus, that allows communication with other ρ-VEX processors, peripherals and the host machine. AHB is an Advanced Microcontroller Bus Architecture (AMBA)-based, open standard, on-chip bus developed by ARM. A debug bus allows the host machine to keep track of about everything that's going on inside the ρ-VEX processor.

2.2.2.4 Memory hierarchy

As stated in Section 2.2.2.2, each of the ρ-VEX cores implements a general register file. This register file is the lowest layer in the memory hierarchy, closest to the ρ-VEX core. It is used for storing intermediate results of operations and other information vital for the working of the core.

The second layer in the memory hierarchy is formed by either cache memories or regular, single-port memories, depending on the preferences of the user. The ρ-VEX supports cache memory as intrinsic parts of the ρ-VEX system. The cache consists of a separate data and instruction cache, both of which are reconfigurable in terms of size, associativity and replacement policy. However, the caches are optional components of the system and can be turned off at design-time, in which case they are replaced by a regular, single-port data and instruction memories.

Dynamic Random Access Memory (DRAM) forms the third layer of the memory hierarchy. This memory is located off-chip, but on the FPGA development board. It

is significantly slower than the on-chip caches and memories, but accessing it is much faster than accessing memory on the host machine.

The highest layer in the memory hierarchy is the memory located on the host machine, and is in most cases represented by a hard disk drive. Accessing this memory is extremely time consuming and costly, because of three reasons. Firstly, the paths from the ρ-VEX core to this memory are long, making communication very time consuming. Secondly, the hard disk drive is relatively slow as compared to the others memory types. Thirdly and finally, while data is being fetched from the host machine's hard drive, the ρ-VEX processor is stalled, resulting in wasted time.

2.2.3 Instructions

Instructions for VLIW processors are more complex than instructions for conventional processors. To optimally exploit ILP, the VLIW processor should be able to process multiple operations in parallel. To accommodate this, two layers of abstraction are added to the instruction. A VLIW instruction is a collection of so-called *syllables*. Each syllable contains an *operation*. By using this model, the VLIW processor is capable of parallel processing, even though it can only process one instruction at a time. VEX operations are semantics-based and format-independent, so the same opcode can be used regardless of the type of operands and the results.

2.2.3.1 ISA

As explained in Section 2.2.2.2, the ρ-VEX processor implements the VEX ISA as introduced by Joseph Fisher. The term ISA is somewhat confusing in this sense, since it assumes a single instruction to correspond with a single operation. Since the ρ-VEX is a VLIW processor, this is not the case. As explained in the previous section, a single instruction can contain a collection of operations. To avoid confusion, the contents of the VEX ISA are referred to as operations rather than instructions. A complete overview of all operations enlisted in the VEX ISA are listed in Table 2.2 and Table 2.3. A discussion on the contents of this ISA can be found in Section 2.4.3, where the VEX ISA and the ISA of the GPU are compared.

2.2.3.2 Scheduling and execution

The scheduling of operations on a VLIW platform is relatively complex. The scheduler needs to order the instruction in such a way that the ILP is exploited optimally. To allow for this the scheduler should be able to issue multiple operations simultaneously and issue them out of order if necessary. When multiple operations are scheduled for simultaneous execution, they should be independent of each other. If they do depend on each other, the processor will experience race conditions during execution, which might cripple the working of the processor or corrupt the data that is being processed. If the scheduler can't schedule enough operations to fill the entire width of the ρ-VEX processor, a remaining slot is filled with a No Operation (NOP).

Because this way of scheduling is complex, it requires a relatively large amount of resources and time. Because of this, scheduling is handled by the host. Performing the

scheduling on the FPGA would be a waste of precious resources. When the host machine handles the scheduling, the VLIW processor is able to make optimal use of the FPGA resources. On top of that, it makes sense to let the host machine handle scheduling, since it is being performed during compilation of the source code, which is also performed by the host CPU.

2.3 GPU analysis

The GPU is an indispensable component of a modern computer system. GPUs are designed especially to handle all forms of image processing, but have also proven to be extremely powerful hardware accelerators. The core assets of the GPU are its impressive parallel processing capabilities [1]. What also makes GPUs attractive is the fact that they are relatively cheap, easy to program and applicable in a broad application domain that stretches way beyond image processing. This section aims to explain how the GPU works, and what enables it to process massive amounts of data in parallel.

2.3.1 Parallelism

The GPU is a parallel processor, that exploits various sorts of parallelism. This section introduces the types of parallelism exploited by the GPU. There are several types of parallelism known in computing, and the GPU exploits nearly all of them. These include ILP, Data-level Parallelism (DLP) and Thread-level Parallelism (TLP). ILP is already introduced in Section 2.2.1. DLP is exploited by performing a single operation to multiple subsets of data. This increases the amount of data that is processed. Also, the scheduling and instruction hardware is not increased in size or complexity. TLP is exploited by performing different operations simultaneously in order to fully utilize the available resources.

2.3.2 GPU design

Unlike a VLIW processor, or any other type of CPU, a computation accelerators like the GPU is never used independently, but is always part of a larger computing system. Even though it is used as an extension of a host machine, the GPU is equipped with its own memory and processor. In most cases, the GPU is connected to the other main components of the computing system via a high speed bus, like PCIe. Figure 2.2 illustrates a common configuration of computer system featuring a GPU.

2.3.2.1 Processor architecture

The parallel processor of the GPU is designed to process large quantities of data as fast as possible. To achieve this, GPUs handle data in a stream. This means that new data is transferred from and to the processor in a stream, whenever it's needed. This reduces the requirements for costly data storage on the chip itself. The processor needs to have a high data throughput to process the continuous stream of data. To achieve this the processor consists of a large amount of independent processing elements, called Streaming Processor (SP) cores, that work together to process the vast amounts of data that are fed

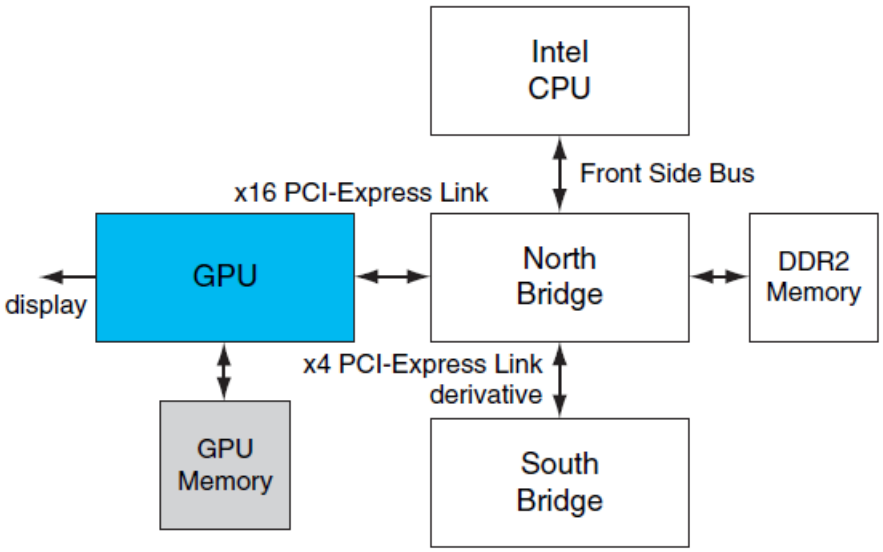


Figure 2.2: Computer system with an Intel CPU, adapted from [1]

to the processor. An SP core is a 32-bit floating point processor. These SP cores need to be structured efficiently so they can be targeted easily. A tree-like structure allows communication between the highest and lowest architectural layers in just a few steps, making it effective in exploiting parallelism. A structure like this can be implemented by introducing layers of abstraction to the architecture. A typical modern hierarchy entails several hundred SP cores, divided over a few dozen groups, known as Streaming Multiprocessor (SM) cores. If the leaves of the tree-like structure are represented by the SP core, the branches are represented by these SM cores. The SM cores are able to operate in an Single Instruction Multiple Data (SIMD)[2]-like mode. This mode, called Single Instruction Multiple Threads (SIMT) in GPUs, allows control over multiple SP cores by a single instruction source. This diminishes scheduling and operating overhead, and makes parallel execution less complex. Besides approximately eight SP cores, each SM core also contains two dedicated Special Function Unit (SFU) cores and two double precision units, that aid the SP cores with performing complex operations. To allow the SP cores to work the most effectively, each SM core is divided into two so-called SP lanes, each of which has control over half of the resources of the SM core. Sometimes two or four SM cores are entailed within a so-called Texture/Processor Cluster (TPC), which provides an additional layer in the hierarchy to further increase the parallel processing capabilities of the structure.

A graphical representation of a typical modern architecture is shown in Figure 2.3.

2.3.2.2 Memory architecture

Thanks to its architecture, the parallel processor is able to process incredible amounts of data in parallel. However, in order to able to benefit from this processing power the

[2]SIMD is a method for exploiting DLP, by applying a single instruction to multiple pieces of data.

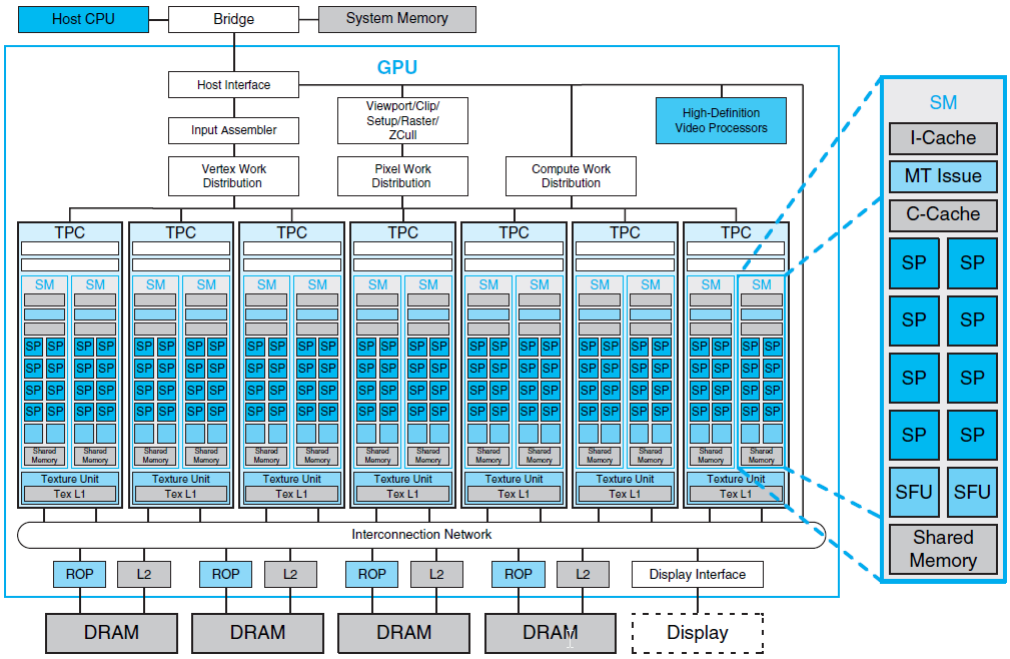


Figure 2.3: Basic GPU architecture, adapted from [1]

processor needs to have a way to provide all of its cores with data to work on. This requires a hierarchical and parallel memory structure much like the processor hierarchy. To this end, the GPU memory architecture has a few important characteristics. First of all the memory banks used in the GPU are very wide, providing high data bandwidth. The memory is fast, thanks to aggressive signaling techniques that improve the transfer speed of the data. Also data compression techniques are used, to minimize the amount of data that has to be transfered. Finally structures like caches and scratch-pad memories are used to reduce the need for fetching data from outside the chip. Fundamentally, high throughput is in conflict with low latency, and thus the GPU has to deal with relatively long waiting times whenever off-chip data needs to be fetched. Luckily this does not have to be a problem. The massive parallelism of the GPU can hide this latency, by executing other tasks from other cores whenever some core is waiting for data.

Memory types The GPU uses several types of memory in the layered structure, each with its own properties and purposes.

The highest level is the *global memory*, which is shared by the entire GPU and can be accessed by all cores. It can even be accessed by the CPU of the computer system. It is meant for communications among the cores and for storage of larger amounts of data. It is implemented in off-chip DRAM. Transferring data from and to this DRAM is relatively the most time consuming on-chip memory operation for the GPU.

The level below the global memory is the *shared memory*, which is exclusive to an SM core. This shared memory is instantiated whenever it is needed and deleted when it is no longer needed. This dynamic behavior ensures that the shared memory does not need

to be large, allowing it to reside in the on-chip Static Random Access Memory (SRAM) cells. Fetching from this type of memory takes significantly less time, due to the location and the nature of the Random Access Memory (RAM) blocks in which it is implemented. It is mainly used for communication among the SP cores that are working on the same task.

The level below the shared memory is the *private memory*, which is specific to a single SP core. Each of these is a assigned a specific section of off-chip DRAM on the GPU. This memory is used for large portions of temporary data, such as private variables that are too large for the registers of the SP core. Since accessing the off-chip DRAM is very time consuming, modern GPU systems tend to cache portions of the private memory on-chip.

The lowest level are the *registers* of the SP core. These are located in the SP core itself. While the amount of data that can be stored in the registers is extremely small as compared to the other memory types, they are vital to the working of the SP core.

Constant memory spaces Besides the three aforementioned memory spaces there are two special read-only memory spaces. The first is called the *constant memory*. This memory is located in off-chip DRAM and can be filled at compile time to the liking of the programmer. Filling the constant memory with data that is needed during execution, ensures that the GPU will have less need to fetch data from the host machine's memory, thus improving the performance of the GPU. The second constant memory space is the *texture memory*, which holds large arrays of data and it also located in off-chip DRAM. This memory can be used in the same way as the constant memory, but it is optimized for spatial locality. This means that whenever data is placed in the texture memory efficiently (e.g. with unit stride), subsequent pieces of data can be transferred to the processor efficiently. Data from both the constant and the texture memory can be cached on-chip.

2.3.3 Instructions

The GPU has it's own dedicated ISA that is able to exploit the parallel capabilities of processor. This section will discuss the contents of one particular GPU ISA and the instruction scheduling procedure of the GPU.

2.3.3.1 ISA

Like every other processor, the GPU features a certain set of instructions that it can executed. Commercial ISAs are often proprietary, because in disclosing these their respective owners might give away vital information that could distort market competition. For this reason, GPU manufacturers add an abstraction layer on top of their ISAs to hide their proprietary information, while allowing the programmer to interface with the GPU. NVIDIA, one of the largest GPU manufacturing companies calls the abstraction layer of their ISA Parallel Thread Execution (PTX). NVIDIA supplies thorough documentation of PTX, giving us a good impression of the GPUs ISA [23]. PTX is also covered in detail in a widely used computer architecture book by Patterson and Hennessy [1].

A detailed overview of the instructions included in the PTX ISA can be found in Section 2.4.3, where it is compared to the VEX ISA.

2.3.3.2 Scheduling and execution

With the processor and memory architecture fine-tuned for high speed parallel processing and the instructions available for doing so, the final challenge is in scheduling these instruction in the best way possible. The scheduler does not target individual cores, but targets the layers of the parallel processor. It uses multiple schedulers to do this. This section aims to explain why and how the GPU does this.

Targeting the GPU As mentioned in the beginning of Section 2.3, the GPU always functions as an accelerator and never in a stand-alone environment. To use the GPU, it must be targeted via a host machine. To make targeting the GPU easier, multiple programming frameworks have been introduced. Best known is Compute Unified Device Architecture (CUDA), which is introduced by GPU manufacturer NVIDIA itself [24]. Another widely used framework that can be used to target GPUs is Open Computing Language (OpenCL) [25]. Frameworks like these allow programmers to easily run workloads in parallel on a GPU. Obviously the programmer has to bear in mind how the instructions will be scheduled on the GPU and how data is addressed by the GPU while writing a parallel program, to gain optimal performance.

Scheduling hierarchy Scheduling of individual instructions to individual cores is not a feasible option for the GPU. The large amount of individual cores would make this option very time consuming. For this reason, instructions are scheduled in a hierarchy that maps precisely to the architecture of the parallel processor.

The highest layer in this hierarchy is the so-called *Grid*. This is a vectorizable loop that contains several smaller *Thread Blocks*, which form the second layer in this hierarchy. The main idea is that a Grid is mapped to a TPC or SM core and the Thread Block is mapped to an SP lane. The lowest level in the hierarchy is located inside the Thread Block and is called a *Thread*. A Thread is mapped to an SP core, and is simply a sequence of PTX instructions. Using a SIMT based approach, it is also possible to target multiple SP cores with a single instruction. This structure enables the parallel processor to easily issue a vast amount of instructions with minimal scheduling overhead.

Figure 2.4 shows the scheduling hierarchy along with the various types of memory that are accessible by each layer, as introduced in Section 2.3.2.2.

Drawbacks However, there are obviously some downsides to this way of scheduling.

For instance a Thread is not independent, since it is only able to perform the same task as every other Thread in the same Thread Block. It is for this reason that GPUs only perform well for data-level parallel problems. Since our target application entails these kind of problems, this is of no concern.

Also scalar execution cannot be done efficiently given this scheduling hierarchy. In a worse case scenario the GPU has to disable all but one SP cores in a lane to execute

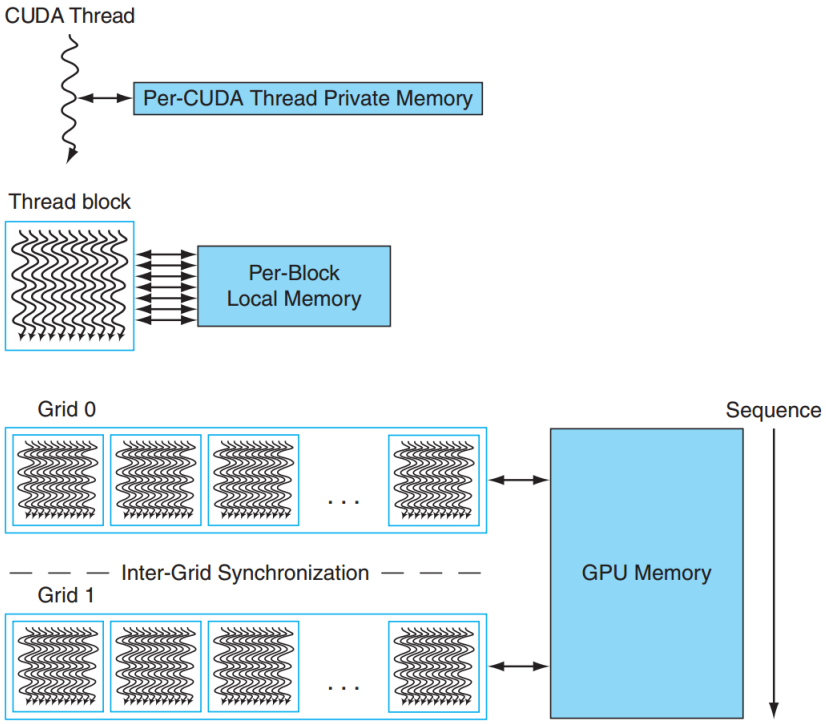


Figure 2.4: Scheduling hierarchy in GPU system, with the accessible memory per layer [1]

instructions on scalar data. Even though this is a solution that causes enormous performance loss, it is the best way to handle this kind of problems. The only other solution would be to outsource such problems to the CPU of the host system, but the overhead of the data transfer makes this option even less attractive.

Obviously the best performance is achieved when all threads follow the same path. Whenever a branch occurs for some of the threads, the performance is decreased immediately. To cope with this, the processor has a special branch synchronization stack that manages threads that diverge and converge. Even so, branching is still the cause of a lot of performance decrease in all sorts of processors, both parallel and others.

2.4 Capability analysis

The design presented in this thesis will aim to combine the benefits of both the GPU design philosophy and the ρ-VEX VLIW processor in a single computation fabric, using the ρ-VEX as a starting point. To create an effective design the useless parts of the ρ-VEX will have to be identified and removed, and the useful parts and properties of the GPU will have to be ported to the FPGA based setting, and implemented in the computation fabric.

2.4.1 Architecture

The structure of the GPU is created specifically for image processing applications much like our target application, while the ρ-VEX processor is created for more general purpose tasks. Thus it makes sense to adapt the architecture of the GPU. An architecture that consists of multiple individual processor cores, that are able to handle data in a stream. However, the number of processor cores that can be fit on an FPGA is significantly smaller than the number of cores available on a GPU. Where the GPU can contain over a thousand individual cores, the FPGA only has the capacity for about a hundred at maximum. This means that massive multithreading on a single FPGA is not a realistic option. However, other techniques to exploit parallelism can be implemented on FPGAs. Pipelining is one of these techniques. Several cores can be connected to each other, and form a processor pipeline. When each processor in this pipeline is dedicated to a task, multiple operations can be performed on data that is being processed by this pipeline. This increases the throughput of the processor, at the cost of latency.

2.4.2 Memory architecture

The memory hierarchy of our fabric should be able to support the high throughput of the processor pipeline. This does not require many different memories in terms of size and speed, but it requires fast and well-placed memories width a high bandwidth. Caching techniques are not required, since data is never being reused in image processing tasks. Each pixels is only being handled once. Placing the memories efficiently and ensuring single-cycle access will minimize their impact on the critical path of the fabric. The critical path is defined as the longest possible path through the design. In order to be able to meet the timing requirements, the design must be able to traverse this path within a single clock cycle. The longer this path, the longer a clock cycle needs to take, causing a lower operating frequency of the overall design. By structuring the memory hierarchy in the same way as the processor pipeline, the data can be streamed through the processor pipeline. This memory architecture effectively removes the latencies related to fetching data from off-chip memories that would occur in a pipeline-less design. The latencies can be hidden entirely by implementing a technique like Direct Memory Access (DMA), that enables a system to interface with external memories without the interference of a processor. This can also be done in a streaming fashion, effectively making the fabric a low-latency streaming multiprocessor.

2.4.3 Instruction Set Architecture

The operations and instructions from the VEX and PTX ISAs are listed in Table 2.2 and Table 2.3. The two ISAs are shortly examined and compared in terms of usefulness and applicability in our fabric. The instructions of both ISAs can be categorized in five groups.

Arithmetic instruction The VEX ISA has fewer traditional arithmetic instructions that the PTX ISA, but implements some exotic functions, like shift-and-add and sign operation.

Multiplication instructions While the PTX ISA provides a single instruction for multiplications, the VEX ISA has a rich set of dedicated operations for this. These are needed to be able to handle data that varies in terms of size and signedness, and are implemented to support Digital Signal Processing (DSP) tasks.

What also stands out is the *multiply-add* instruction in the PTX ISA, which combines a multiplication and an addition in a single instruction. In image processing, which is the core task of a GPU, a multiplication and an addition in sequence are quite common. When trying to gain maximal speed and performance, it is important to make the common case tasks fast. The multiply-add instructions aims to do just this.

The VEX ISA only supports division steps as an operation. This means that dividing larger numbers using the VEX ISA will require a lot of these operations, making it quite costly.

Special functions instructions This group only holds instructions of the PTX ISA, which are executed by the aforementioned SFU core of the GPU. Among these are functions like the sine, the cosine, the logarithm and the square root. Again, these functions are implemented to make the common case fast. The VEX ISA does not support any special functions.

Logical functions instructions This group contains all logical and binary operations, like AND, OR, XOR and bit-shift operations. The VEX ISA supports binary and conditional implementations of most of these operations, while the PTX ISA only supports conditional operations.

Memory access instructions This group lists all memory related instructions, like load and store. These instructions provide communication between the processor and memory. Again the PTX ISA implements just the needed basic operations, and the VEX ISA has multiple operations to support different data sizes and signedness.

Control flow instructions Among these are regular instructions like instruction calls and branches, and the PTX ISA also has an atomic instruction. This instruction makes sure that the parallel tasks can be synchronized. Thanks to instructions like these, parallel execution op GPUs can be kept uncluttered. The control flow instructions of the VEX ISA are again rather large in number and allow for flexibility in the execution of programs on the ρ-VEX system. Since atomic execution of operations embedded in an instruction is an intrinsic part of the VLIW design philosophy there is no dedicated atomic operation in the VEX ISA.

Conclusion There is a large overlap between the two, so they are theoretically both suitable. However, since the desired starting point is the ρ-VEX, it makes sense to start out with the VEX ISA. There are two major differences between the two ISAs. Firstly, the VEX ISA only supports signed and unsigned integers as data type, while PTX supports 8-, 16-, 32- and 64-bit signed and unsigned integers, untyped bits and floating point signals. This makes PTX intrinsically more powerful, but also more exhaustive

in terms of scheduling and resources. Secondly, the VEX ISA lacks support for special functions, like square root or sine operations. These are beneficial when hard mathematical workloads are executed. Also the multiply-add instruction from the PTX ISA might be beneficial when it comes to performing image processing algorithms.

Table 2.2: VEX and PTX overview - Part 1

Instruction class	VEX Instruction	PTX Instruction	Instruction description
Arithmetic	ADD	ADD	Add
	ADDCG		Add with carry and generate carry
	SUB	SUB	Subtract
		REM	Remainder
		ABS	Absolute value
		NEG	Negative value
	MIN	MIN	Minimum
	MINU		Minimum unsigned
	MAX	MAX	Maximum
	MAXU		Maximum unsigned
	SH1ADD		Shift left one and add
	SH2ADD		Shift left two and add
	SH3ADD		Shift left three and add
	SH4ADD		Shift left four and add
	SXTB		Sign extend byte
	SXTH		Sign extend half
	ZXTB		Zero extend byte
	ZXTH		Zero extend half
		MOV	Move
		CVT.DTYPE	Type conversion
Multiplication		MUL	Multiply
	MPYLL		Multiply signed low 16 x low 16 bits
	MPYLLU		Multiply unsigned low 16 x low 16 bits
	MPYLH		Multiply signed low 16 x high 16 bits
	MPYLHU		Multiply unsigned low 16 x high 16 bits
	MPYHH		Multiply signed high 16 x high 16 bits
	MPYHHU		Multiply unsigned high 16 x high 16 bits
	MPYL		Multiply signed low 16 x 32 bits
	MPYLU		Multiply unsigned low 16 x 32 bits
	MPYH		Multiply signed high 16 x 32 bits
	MPYHU		Multiply unsigned high 16 x 32 bits
	MPYHS		Multiply signed high 16 x 32 bits and shift left 16 bits
		MADD	Multply and add
		DIV	Divide
	DIVS		Divide step
Special functions		RCP	Reciprocal
		SQRT	Square root
		RSQRT	Reciprocal square root
		SIN	Sine
		COS	Cosine
		LG2	Binary logarithm
		EX2	Binary exponentional

Table 2.3: VEX and PTX overview - Part 2

Instruction class	VEX Instruction	PTX Instruction	Instruction description
Logic	AND		Bitwise AND
	ANDC		Bitwise Complement and ADD
	OR		Bitwise OR
	ORC		Bitwise Complement and OR
	XOR		Bitwise exclusive OR
		AND	Conditional AND
	ORL	OR	Conditional OR
		XOR	Conditional exclusive OR
	NANDL		Conditional NAND
	NORL		Conditional NOR
		NOT	One's complement
		CNOT	C logical not
	SHL	SHL	Shift left
		SHR	Shift right
	SHR		Shift right signed
	SHRU		Shift right unsigned
	CPM		Compare
		SETP.CMP	Compare and set predicate
		SELP	Select with predicate
	SLCT		Select if true
	SLCTF		Select if false
Memory access	LDW	LD.SPACE	Load word
	LDH		Load halfword signed
	LDHU		Load halfword unsigned
	LDB		Load byte signed
	LDBU		Load byte unsigned
	STW	ST.SPACE	Store word
	STH		Store halfword
	STB		Store byte
		TEX.ND.DTYP	Texture lookup
		ATOM.SPC.OP	Atomic read-modify-write
	PFT		Prefetch
Control flow	GOTO		Unconditional relative jump
	IGOTO		Unconditional absolute indirect jump
	CALL	CALL	Call function
	ICALL		Unconditional absolute indirect call
	BR	BRANCH	Conditional branch
	BRF		Conditional relative branch on false
	RETURN	RET	Return from function call
	RFI		Return from interupt
	XNOP n		Multicycle NOPs
		BAR.SYNC	Barrier synchronization
		EXIT	Terminate thread execution

2.4.4 Scheduling

Scheduling for the ρ-VEX is complex, but does help to exploit ILP. This is an important feature that is beneficial for the performance of our computation fabric. Besides

the complexity, scheduling for the ρ-VEX has no significant downsides. The SIMT approach of the GPU scheduler might be interesting for our computation fabric, since it decreases the complexity of scheduling. Incorporating this approach into the design of our computation fabric might improve its performance.

3 Image processing algorithms

In order to dictate the requirements for our computation fabric, it is needed to understand what workloads it will be expected to process. The primary function of the fabric will be to perform all functionalities required for it target application; medical imaging. This chapter aims to analyze the complexity of this application, and deduce the requirements posed by it. Medical image processing, and image processing in general, is mostly performed by applying a number of algorithms to an image, altering the contents of the image to the needs of the user. To indicate the complexity of image processing algorithm it is necessary to understand the basics of algorithm analysis. The first part of this chapter will thus discuss basic algorithm analysis. Secondly, an overview of image processing algorithm classes is discussed, along with a short discussion of few example algorithms from each class. Finally a medical image processing algorithm workload is constructed using a number of these example algorithms. The requirements posed by this algorithm workload are used in the design of our computation fabric.

3.1 Analyzing algorithms

Describing the complexity of algorithms is a basic task in computer science. The complexity is expressed in terms of required resources, such as execution time and memory space, to execute one or more algorithms. Most often a complexity analysis is performed when multiple algorithms need to be compared, in order to see which one is most suited to perform the target application on a given platform. However, in case of this research it is the other way around. The target application and algorithms are known, and the platform should be optimized for this. The complexity of these algorithms needs to be analyzed to describe the requirements of the computation fabric that has to be implemented.

3.1.1 Basic metrics and analysis

In order to be able to analyze algorithms, certain standards and metrics have been introduced. Algorithms are mostly analyzed asymptotically to describe their complexity and behavior when large sets of input data are involved. The Bachmann-Landau notation helps to express the complexity asymptotically [26]. This family of operators describes various kinds of bounds on asymptotic growth rates. Table 3.1 shows the relevant operators for algorithm analysis from this family, along with their meaning. Out of these, the Big O notation is the most useful for the purposes of this analysis, since it discusses the upper bound of computational complexity for large data sets. Images for our target application consist of several thousand or million pixels, they can be regarded as large sets. Note that the asymptotic upper bound is always expressed as $O(n^x)$, where n is

the size of the data set.

Table 3.1: Relevant selection of operators from Bachmann-Landau family

Operator	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$	$f(n) = \Theta(g(n))$
Pronunciation	Big O	Big Omega	Big Theta
Definition	$\|f(n)\| \leq k \cdot \|g(n)\|$ for some positive k	$f(n) \geq k \cdot g(n)$ for some positive k	$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ for some positive k_1, k_2
Explanation	f is bounded above by g asymptotically	f is bounded below by g asymptotically	f is bounded both above and below by g asymptotically

Algorithm analysis is usually limited to determining the complexities of required time and memory. The Bubble sort algorithm will be used to demonstrate how these three complexities can be found. Pseudocode for this algorithm is in Algorithm 1.

Algorithm 1 Sequential implementation of the bubble sort algorithm

```
1:  procedure BUBBLE SORT(a[n])          ▷ Produces a sorted list with n elements
2:      for i from 1 to n do
3:          for j from 0 to n do
4:              if a[j] > a[j + 1] then
5:                  temp ← a[j]
6:                  a[j + 1] ← a[j]
7:                  (a[j] ← temp
8:              end if
9:          end for
10:     end for
11: end procedure
```

The bubble sort algorithm takes array a that consists of n elements as input. It compares two elements of the array at a time, and works from the first until the last element. Whenever the first element is larger than the second, the two are swapped. It then continues to compare the second and the third element, and so on. In this way the largest number in the array will end up at the end of the array when after one iteration. At this point a total of $n - 1$ comparisons have been performed. The algorithm continues to iteratively sort the array in this fashion until all elements are in the correct place. The result is a sorted array, in which a following element is never smaller than its predecessor. Note that there is an easy and obvious optimization for this algorithm, which is not applied here in order to keep the example as simple as possible. It is possible to reduce the number of comparisons by one every iteration, since each iteration the highest encountered element will be shifted to the end of the array and thus does not need to be compared anymore.

Time complexity To find the asymptotic bounds of required execution time, the easiest way is to determine the order of the amount of operations that need to be performed by the algorithm when working on a dataset of size n. The asymptotic bound indicates a worst case performance. It might be so that there is no possible input for the algorithm that would ever realize this worst case situation. Therefor, an analysis that looks for these sort of bounds is also referred to as pessimistic. Pessimistic analysis will indicate high constraints and requirements, but whenever these are lived up to, there is no chance for the system to fail in the execution of this algorithm. The bubble sort algorithm consists of two nested for loops that surround an if-statement. In a worst case situation both for loops are executed wholly, along with the if-statement. This means that a total of $n-1 \cdot n \cdot 1$ operations will be performed. In Bachman's Big O notation this means that the algorithm has a worst case time complexity of $O((n-1) \cdot n \cdot 1) = O(n \cdot n \cdot 1) = O(n^2)$.

Memory complexity This metric indicates the highest amount of memory that is allocated by the algorithm at a single time. The bubble sort algorithm performs a single comparison at a time. So at most it operates on two pieces of data simultaneously, but never more. For this reason the worst case memory complexity for the bubble sort algorithm is of order $O(2) = O(n^0) = O(1)$, since it considers two elements of the array during a comparison.

3.1.2 Analysis for parallel algorithm execution

Up to this point the analysis assumed a single core system that sequentially executes the algorithm. However, when a multicore computing system is considered, the complexities change. Again the bubble sort algorithm is used as example, but this time it is adapted for parallel execution. Parallel execution can be implemented by many techniques, such loop parallelization and divide and conquer. The goal of all parallelization approaches is to divide the total work into smaller chunks that can be processed individually. One of the most basic techniques for parallel execution is parallelizing the loops of the algorithm. Of course this is only possible if no dependencies exist between the iterations of loops. We apply this approach to the bubble sort algorithm. The outer loop of the the bubble sort algorithm can not be parallelized, since each following iteration is dependent on the previous one. Executing steps of the outer loop in parallel or out of order would falsify the outcome of the algorithm. The inner for loop of the algorithm can be parallelized, but a trick is needed. Since a comparison checks and possibly swaps two elements at a time, the multiple cores cannot work on neighboring elements. For this reason, each core will be assigned two unique pieces of neighboring data. In the even iterations of the outer loop, the inner loop will compare and possibly swap the first and second, the third and fourth, the fifth and sixth etc. pieces of data. In the uneven steps the second and third, fourth and fifth, sixth and seventh etc. array elements are handled. This approach is called odd-even transposition. Pseudocode for this parallel algorithm is shown Algorithm 2. The parallelized for loop is indicated by *parfor*.

Now it is clear how the bubble sort algorithm can be parallelized, its complexities can be determined.

Algorithm 2 Parallel implementation of the bubble sort algorithm

```
1: procedure BUBBLE SORT(a[n])                ▷ Produces a sorted list with n elements
2:     for i from 1 to n do
3:         parfor j in 0 .. 2 .. 4 .. [..] .. n do
4:             if i % 2 = 0 then
5:                 if a[j] > a[j + 1] then
6:                     temp ← a[j]
7:                     a[j + 1] ← a[j]
8:                     (a[j] ← temp
9:                 end if
10:            end if
11:            if i % 2 = 1 then
12:                if a[j + 1] > a[j + 2] then
13:                    temp ← a[j]
14:                    a[j + 1] ← a[j]
15:                    (a[j] ← temp
16:                end if
17:            end if
18:        end parfor
19:    end for
20: end procedure
```

Time complexity Parallel computing was introduced as a measure to reduce execution time om workloads, so obviously time complexity is an important metric in parallel computing. Determining the time complexity is not very different in parallel algorithm analysis as compared to sequential algorithm analysis. Given that the inner loop of the bubble sort algorithm can be handled in parallel as described in the previous section, it can be executed in $O(1)$ time given that a sufficient number of processors is available. This complexity is determined by dividing the size n of the data set by a the maximum number of $n/2$ processors that can operate on the data at the same time. This results in a complexity of $O(\frac{n/2}{n}) = O(1/2) = O(1)$. However, the iterations of the outer loop are dependent of each other and cannot be performed in parallel, leaving the time complexity of the outer loop at $O(n)$. The total time complexity of the parallel algorithm comes down to $O(1) \cdot O(n) = O(n)$.

Number of processors Parallelization poses a trade-off between execution time and the number of used resources. The number of processors used for parallelization varies between two and the logical upper bound posed by the algorithm. For bubble sort the maximum number of needed cores is $(n/2)$, since each iteration only half of the array can be used for computations. For this reason the number of processor cores is of order $O(n/2) = O(n)$.

Total cost The combination of time complexity and the number of used processor cores makes up the total cost of the parallel algorithm. It is defined as Total cost =

Time complexity · Number of processors. In this case this comes down to Total cost = $O(n) \cdot O(n) = O(n^2)$.

Speedup and efficiency In order to determine if a parallel algorithm is efficient, the speedup per processor is the principal measure. Speedup is defined as the sequential time complexity divided by the parallel time complexity. For this example the speedup is of order $O(n^2)/O(n) = O(n)$. Whenever the order of the speedup is equal to or greater than the order of the number of processors, the algorithm is said to be scalable. The efficiency of a parallel algorithm is defined as the speedup over the number of processors used. The efficiency of the bubble sort algorithm is $O(n)/O(n) = O(1)$, which indicates that it is viable for parallelization.

3.2 General image processing algorithms overview

The processing of digital images is relative compute intensive and is performed by special image processing algorithms. This section will first give a short introduction on the various classes of image processing algorithms. Then the possibilities of parallelizing image processing algorithms are shortly discussed.

3.2.1 Image processing algorithm classes and types

Image processing entails quite a broad spectrum of operations on visual data. Image processing is applied in numerous fields throughout our entire society and industry. Because of this wide application domain it is hard to give an overview of all types of algorithms. Therefore, a selection of algorithms is presented that is sufficiently representative for the challenges posed by the medical imaging target application. A few of these algorithms will be used to compose an algorithm workload for our computation fabric. The algorithms are divided in a number of basic classes. Note that all algorithms operate on an image in the same way. Images are processed from left to right, and from top to bottom. So the images are processed in lines.

Image scaling Algorithms in this class are able to resize an image by either shrinking or enlarging it. The most common way of doing this is called interpolation. Whenever an image needs to be enlarged the number of pixels in the image increases. The values of the new pixels are determined via a technique called interpolation. A new pixel is placed between two or more existing pixels and is assigned a value based on the already existing neighboring pixels. When scaling an image down, a selection of existing pixels is removed.

Nearest Neighbor interpolation This is a very naive interpolation algorithm. Whenever a new pixel needs to be added via interpolation, it is simply assigned the value of the nearest neighbor. No averaging or other calculations are performed.

Bi-linear interpolation Contrary to the nearest neighbor variant, this algorithm is not naive at all and is computationally intensive. Pixel values in both horizontal and vertical direction are used to calculate new pixel values whenever new pixels are introduced by interpolation.

Color correction and conversion Enhancing the color or contrast of images helps to improve the overall quality of the image. It can also be used to remove or alter color information of an image. An example algorithm class that removes color information, and thus converts images to grayscale format, is grayscaling. An example that enhances color information is histogram equalization. Both of these will be discussed shortly.

Gray scaling Gray scaling algorithms take the coloring information of an image and use that to determine the weight of the shade of gray that needs to be used. The calculation that determines the appropriate shade can be implemented in numerous ways, varying from easy to relatively complex. In all cases, gray scaling algorithms operate on a single pixel at a time, resulting in a relatively low complexity.

Histogram equalization A histogram of the image is created, indicating the pixel values in an image and the frequency of their occurrence. This results in a distribution that in most cases does not cover the entire available color spectrum. This algorithm stretches the distribution over the entire spectrum to relatively increase the difference between the pixel values thus increasing the color contrast in the image. This algorithm needs to process the image two times. A first time to generate the histogram and a second time to update all pixel values, making it complexer than the gray scaling algorithm.

Spatial and temporal filtering Filtering techniques are often used in pre-processing steps of image processing in order to remove noise or other unwanted artifacts from the source images in order to improve the working of algorithms that need to extract data from the image. They are very compute intensive, since they operate in a *frame-based* manner. This means that they do not consider on a single pixel at a time, but consider a *frame* or *window* of pixels surrounding the pixel that is being operated on. This frame is typically has dimensions of $3x3$ or $5x5$ pixels. A *filter* of the same size is applied to this frame. A filter window is always a square with an odd number of pixels on each side. Regarding all the values in the window, the center pixel value is calculated. This way of operation makes filtering a very powerful tool that is capable of many different and complex applications, but does require more time and resources to be performed. More time is required since the number of operations needed to calculate the results is multiplied by the number of pixels in the filter. More resources are required since multiple lines of the images should be considered simultaneously. So when applying a $5x5$ filter to an image, at least five lines of the image should be kept in scope.

Image processing algorithms that apply filtering can be executed more efficiently by applying a so-called *sliding window*. This methods aims to reduce the rate of data fetching by keeping data that is needed in following steps in scope. When a frame based operation is completed, the windows 'slides' to the next pixel and continues to

manipulate it. There is a lot of overlap between the data in the sliding window in consecutive iterations. Each iteration $n^2 - n$ elements of the window can be reused, where n indicates the height and width of the window. A graphical representation of two consecutive steps of an algorithm that uses the sliding window technique is shown in Figure 3.1. The red square indicates the operating window of the frame-based algorithm. The blue rectangle in the second step indicates the data that can be reused, and should be kept in scope for optimal performance.

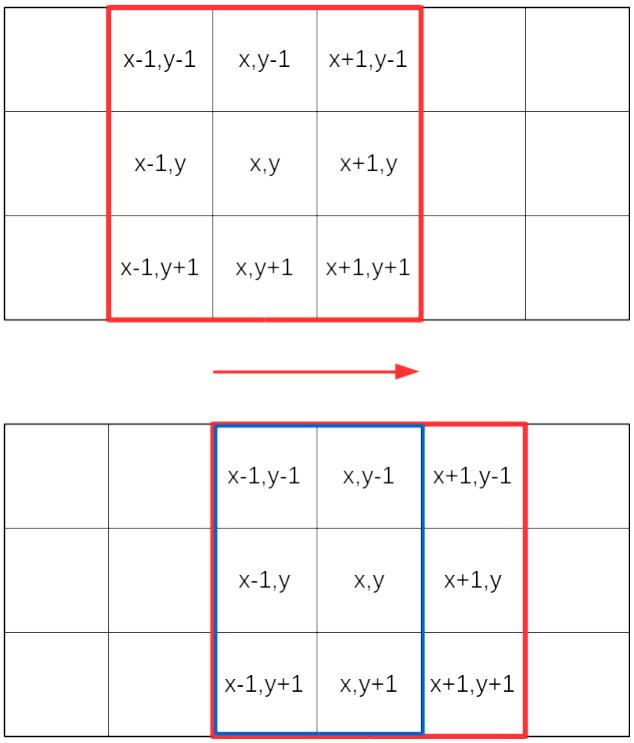


Figure 3.1: Visual impression of two consecutive steps of a frame-based algorithm that uses a sliding window.

Median filter Median filtering replaces each pixel value with the median pixel value of all the pixels in the filter window in order to reduce the noise in the source image. If there are two median pixel values the average of these is used. This algorithm is requires a sorting algorithm to make a distribution of pixels values in the window.

Convolution Convolution is a widely used mathematical operation that modifies a source function by applying a second function to it, thus producing a resulting third function. In case of image processing the source function is the image, the second function is a filter and the third function is the resulting output image. Contrary to the median filter the convolution filter is not fixed and can be tuned to the users liking. This makes the convolution filter extremely flexible and powerful. Practical applications of the this algorithm include sharpening, edge detection, embossing and mean filtering.

Feature extraction Algorithms in this class aim to let the computer discern features in an image. Examples of features than can be extracted from are readable text, faces or shapes. Feature extraction is mostly used to assist humans in interpreting images, but can also be used in computer vision. In computer vision the computer does not only extract information from an image, but also interprets the extracted information itself. Example algorithm subclasses in the feature extraction class are edge detection, thresholding and motion detection algorithms. Two edge detection algorithms are described below. Edge detection algorithms aim to accentuate areas where contrast between neighboring pixels is highest, indicating edges in the image. Like filtering algorithms, edge detection algorithms are usually window-based, making them generally compute intensive.

Sobel edge detection Two 3 by 3 convolution kernels make up the so-called Sobel operator. The combination of these two is used to find the gradient magnitude in two directions at a specific point in a grayscale image. By using two filters, the complexity of this algorithm is relatively high, even higher than the complexity of convolution algorithms.

Canny edge detection This is probably the best known and widespread edge detection algorithm available [27]. It is regarded as highly accurate and very compute intensive, since it implements a five step pipeline of operations including Gaussian filtering, non-maximum suppression and hysteresis. This algorithm is highly accurate, and is also the most complex algorithm of all the examples.

3.2.2 Parallelizing image processing algorithms

The fact that images is build out of individual pixels makes them viable for Data-level Parallelism (DLP). Each pixel is an independent piece of data, making it possible to run pixel based images in parallel easily. However, algorithms that require an entire image at once or need to operate on an image more than once are harder to parallelize. There will be no in depth discussion on how to parallelize this type of images, since the selection of algorithms used in medical imaging applications does not use the types of algorithms, as is described in Section **??**. A third category of algorithms are the window based algorithms, that operate on a small selection of pixels at the same time. It is a bit more tricky to parallelize these algorithms than the pixel based ones, but not at all impossible. When parallelizing this type of algorithms, one should keep in mind that not only the data that is being operated on should be available, but also the surrounding data it.

Also pipelining is an effective technique in parallelizing image processing tasks. However, pipelining is only effective whenever multiple operations have to be performed on the same data, and when the algorithms are of the same computational complexity. Luckily this is the case for medical imaging algorithms, as is explained in the following section. In pipelining, multiple processing stages are chained together. This technique can be implemented in both software and hardware.

3.3 Medical imaging algorithms

A subsection of image processing algorithms is used for the medical imaging. The source of the algorithms used in commercial medical imagers is proprietary software of the respective owners and could not be disclosed even for academic purposes.

3.4 Adaptation of algorithm workload

Based on the outline of the workload in the previous section, an algorithm workload adaptation was created. This section discusses the contents of the algorithm workload and the requirements it poses for our computation fabric.

3.4.1 Contents

The first step in the image processing pipeline is an interpolation algorithm used to scale the size of the source image. The bi-linear and nearest neighbor interpolation algorithms both have the same computational complexity making them equally feasible. Because of its slightly higher flexibility, we select the bi-linear interpolation algorithm for the evaluation. Secondly, a gray scaling algorithm is applied. This algorithm is selected because it operates on single pixels of the input dataset. The third stage is a convolution filter that sharpens the image, followed by the final stage, an embossing convolution filter. The last step of the algorithm workload from the previous section, merging of the resulting images, could not be implemented in time and is thus omitted from our algorithm workload at this point.

3.4.2 Requirement analysis

In this section, the requirements posed to the computation fabric by the compiled algorithm workload are analyzed. First the computational complexities of the algorithms are analyzed. Secondly the size of the programming code that is used to run the algorithm on the computation fabric is profiled.

Computational complexities An overview in terms of complexities for sequential and parallel execution of the image processing algorithms that have been selected for our algorithm workload can be found in Table 3.2. All of these metrics are presented like before, in big O notation. The letter n indicates the number of pixels the image that the algorithm is operating on. This value is typically $960 \cdot 960 = 921,600$ for medical imaging applications. The letter f indicates the number of pixels in the filter that the algorithm is applying to the image. This value is typically $3 \cdot 3 = 9$ or $5 \cdot 5 = 25$, which is significantly smaller than the value of n.

All algorithms roughly have a similar time complexity for sequential and parallel execution. The space complexity is also low, which eases parallelization significantly, since it allows the cores in our fabric to keep all relevant data in their registers and removes the need for large on-chip memories. The algorithms are all equally suitable for parallelization, as is shown by the order of processors that can execute the algorithms

in parallel. This fact is reflected in the statistics on cost, speedup and efficiency. Since the algorithms are roughly equal in all terms, they are perfectly suited to be applied in a pipeline. The convolution algorithm will most likely be the bottleneck in the pipeline due to its slightly higher complexity, caused by the filter it applies. Since the filter size is relatively small, the effects of this bottleneck will be of minor significance.

Since this algorithm workload relies heavily on pipelining, a hardware design that accommodates this is desirable. Our computation fabric should thus implement a pipeline structure of processors, through which the data can stream.

Table 3.2: Complexity analysis of the selected algorithms

Execution model	**Metric**	**Bi-linear interpolation**	**Gray scale**	**Convolution**
Sequential	Time complexity	$O(n)$	$O(n)$	$O(n \cdot f)$
	Space complexity	$O(1)$	$O(1)$	$O(f)$
Parallel	Time complexity	$O(1)$	$O(1)$	$O(f)$
	Processors	$O(n)$	$O(n)$	$O(n)$
	Cost	$O(n)$	$O(n)$	$O(n \cdot f)$
	Speedup	$O(n)$	$O(n)$	$O(n)$
	Efficiency	$O(1)$	$O(1)$	$O(1)$

Code size Also the size of the kernel code of the algorithms in the workload is inspected, along with the additional programming code in which their embedded. The results of this analysis are included in Table 3.3. The algorithms are all close to 1 KiB in size, while the size of the entire program is close to 6 KiB, indicating an overhead of about 2 KiB. Initialization is especially an important factor of this overhead. The large initialization is needed to ensure all cores in the processor pipeline are running when the actual kernels start running.

Since the size of the entire program is around 6 KiB, an instruction cache or memory of at least this size will give optimal results.

Table 3.3: Profiling of created algorithm workload

Component	**Size (KiB)**
Bi-linear interpolation	0.88
Gray scale	0.78
Convolution (Sharpen)	1.03
Convolution (Emboss)	1.00
Loop overhead	0.63
Initialization	1.25
Rest	0.13
Total	5.69

Image lines To support pixel-based and frame-based algorithms, it is desirable to always have a least five lines of an image in scope, as is explained in Section 3.2.1.

Designing the computation fabric

4

This chapter aims to present a design for our computation fabric. First the requirements for the computation fabric formulated in the previous chapters will be summarized. Then the Field Programmable Gate Array (FPGA) platform for design is introduced. This is followed by descriptions of the design of the processor core architecture, the Instruction Set Architecture (ISA) and the memory architecture. Then, a design for the complete computation fabric is introduced. Finally, a design that contains multiple instances of this fabric is introduced. The chapter closes with simulation results of a single instance of the computation fabric.

4.1 Requirements

The previous chapters have led up to a design that combines and incorporates the advantages of the Very Long Instruction Word (VLIW) and Graphics Processing Unit (GPU) design philosophies and is optimized for the targeted medical imaging algorithms in a fashion that allows the exploitation of parallel processing. The following requirements for the design of the computation fabric have been determined.

1. The processor cores in the computation fabric should be
   - based on the ρ-VEX processor
   - as small as possible
   - able to operate in a processor pipeline
2. The computation fabric should
   - be implemented on an FPGA
   - be capable of handling at least five image lines simultaneously
   - have an instruction memory or cache of at least 6 KiB in size

4.2 Platform

Section 1.3. Two FPGA boards were provided by the Delft University of Technology, both of which are also used for the development of the ρ-VEX processor. These are the Virtex-6 FPGA ML605 Evaluation Kit [28] and the Virtex-7 FPGA VC707 Evaluation Kit [29], both developed by Xilinx Inc. An overview of the specifications of these two boards is included in Table 4.1. Block Random Access Memory (BRAM)36 indicates that the size of each BRAM on the FPGAs is 36 kB. The ML605 FPGA board features the Virtex-6 XC6VLX240T-1FFG1156 FPGA and 512 MB of Random Access Memory (RAM). The VC707 features the Virtex-7 XC7VX485T-2FFG1761C FPGA. The

available VC707 does not feature onboard RAM. Even though the VC707 is a newer and better FPGA board, the ML605 FPGA board will be mainly used for the implementation of our computation fabric. The reason for this are the availability of onboard RAM, and the availability of a decent toolchain. The toolchain for development on the VC707 is still in early stages and known to exhibit unstable behavior in certain situations.

Table 4.1: Overview of FPGA specifications

Available resources	**Reg**	**Look-Up Table (LUT)**	**BRAM36**
Virtex-6 XC6VLX240T-1FFG1156	301440	150720	416
Virtex-7 XC7VX485T-2FFG1761C	607200	303600	1180

4.3 Processor architecture

Since the desired platform for the computation fabric is FPGA, it makes sense to take the ρ-VEX processor as a starting point for the design of the computation fabric. The ρ-VEX processor should be modified to optimally exploit parallel processing opportunities. To best support parallel processing, the FPGA should be equipped with as many cores as possible. This requires the cores to be as small as possible. The current design of the full-sized eight-issue ρ-VEX is far from small. The reconfigurability properties and communication between internal components of the processor require excessive amounts of resources. Stripping all unnecessary components from the ρ-VEX will expectedly drastically decrease the resource usage per core. This stripped down ρ-VEX will contain a single two-issue core. This effectively disables all possible internal communication of this core, since it is the only core in the ρ-VEX processor at this point. Reducing the processor size from an issue width of eight to two will expectedly reduce the resource utilization of the processor by a factor of more than four, since three cores and the costly all-to-all communication crossbar are removed from the processor.

Since six of the eight lanes of the ρ-VEX processor are removed in the stripping process, the compute power of the core that remains is a mere quarter of that of the eight-issue ρ-VEX processor. However, the new core is much less complex and will have a shorter critical path. This means that the operating frequency of the core can be increased, compensating for the loss of computing power. While the full-size ρ-VEX is able to operate properly at a maximum frequency of 30 MHz, the stripped core is expected to work at a frequency of 100 MHz or higher. This expectation is based on preliminary synthesis results of ρ-VEX processors that implement a single two-issue core.

Pipeline exploration might bring about ways of increasing the clock frequency even more. By identifying the most time consuming steps in a pipeline and providing them with an addition time slot the critical path can be lowered allowing the operating frequency to be increased at the cost of additional latency caused by the lengthened pipeline. For the ρ-VEX system, and generally for most compute systems, the most time consuming stage is the one where data is being transferred from and to memory. As described earlier, data fetching can require a lot of time. Giving this stage an extra time slot in

the pipeline will improve the timing results of the design, but might also require changes in the compiler or to the design itself.

4.4 Instruction set architecture

Since the ρ-VEX platform is taken as a starting point for the system, the VLIW Example (VEX) ISA is used. This ISA contains all needed instructions for the targeted applications. In the analysis of the VEX and PTX ISAs in Section 2.3.3.1 the lack of the multiply-add instruction and special functions support in the VEX ISA were noted. Adding the special functions to the VEX ISA might be useful for certain use cases, but the algorithms included in the designed algorithm workload from Section 3.4 only use fairly basic operations and would not benefit significantly from these dedicated instructions. However, the multiply-add instruction would be a useful extension of the VEX ISA, since the combination of a multiplication follow by an addition is very common in windows-based image processing algorithms. An obvious example for this is the convolution algorithm, that performs a multiplication on every pixel in the filter window and sums up the results of these in a single variable. Unfortunately, implementing new instructions does require changes to the compiler. This is out of scope for this project, but can be listed as an interesting topic for future improve of the computation fabric.

4.5 Memory architecture

Neither the ρ-VEX or the GPU have a memory architecture that is ideal for our implementation. The memory architecture of the ρ-VEX is too simplistic for our purposes, providing only regular cache memories to each core. Moreover, cache memories are only profitable for applications that reuse data. Image processing algorithms don't generally do this. Apart from the filters used by the image processing algorithms, which are small enough to be stored in the core's registers, no data is ever reused. On the other hand, the architecture of the GPU is too complex for our purposes. Its layered structure would be too costly to implement on an FPGA. Moreover, a memory architecture like that of the GPU is only beneficial in the case of a very large number of processor cores, which is not a reasonable option on the selected FPGAs. The memory architecture in our fabric should be optimized for high throughput, to support the processor pipeline that our computation fabric will implement. This also diminished the needed amount of reads and writes from and to off-chip memory, and allows the data to be streamed through the design. A means to achieve high throughput is by enabling communication between neighboring cores using buffer memories. This method brings about two advantages. Firstly, the bus connecting all the cores and peripherals is greatly relieved when it is not longer used for data transfers, enhancing the working of the fabric overall. Secondly, this buffer memory is not present in either the ρ-VEX or the GPU design, allowing to introduce a new type of memory that can be customized to support the needs of our computation fabric.

4.5.1 Memory type

As explained in the previous section, a buffer memory should be implemented. To optimize this memory type for parallel processing, it should meet a number of conditions. First of all, it should be fast, allowing single cycle access. Secondly, it should feature two ports, making it suited for streaming of data by allowing two cores to interface with it simultaneously. Finally, it should be as small and simple as possible, ensuring a resource usage that is as small as possible. A memory type that satisfies all these conditions is a dual port scratchpad memory [30] [31]. Scratchpad memories are plain memory blocks with no logic or control at all. This makes them small, fast and easy. The downside is that the programmer needs to manually select where the data should be stored in the memory, and whether it is stall valid or not. The two ports allow the memory to be attached to two cores, becoming a part of the streaming processor pipeline.

4.5.2 Memory hierarchy

One of the spear points of the design is high throughput. To ensure this, efficient use of on-chip memory is paramount. This means that enough memory should be available to accommodate streaming sufficient data through the design. Luckily, the ρ-VEX uses relatively more LUTs and flip-flops than BRAMs, which are the needed resources for the local scratch pad memories. It should thus be possible to create an architecture in which the cores and memories can effectively coexist.

To support high throughput, each core in the design should be able to access a memory as fast as possible, preferably in a single cycle. This can only be achieved when a dual port scratch pad memory is connected to each core to function as a *local memory.* It should be possible for a core to access its own local memory and the local memory of a neighboring core, in order to support streaming. When this is not possible, all traffic between the cores must be routed via the bus, which is not capable of handling this much traffic and is slower than interfacing with the local memory directly. With a memory architecture where local memories are connected to the cores directly, the requirement of low latency and high throughput should be guaranteed.

To meet the application's demand it is desirable to be able to store at least five lines[1] of the image at a time. For a typical medical image this would come down to $960 \cdot 5 = 4800$ pixels. Given that each pixel has a size of 16 bits, this image fraction has a size of nearly 10 KiB. Since double buffering[2] is implemented in the fabric, the minimum size of a local memory should be at least 20 KiB. The closest value to this that can be realized is 32 KiB.

Sharing one larger memory with multiple cores was also a considered as a possible approach, and is used in the architecture of the GPU. However, this is not a viable option for our design. Memories with more than two ports require extraordinary amounts of resources, leading up to inefficient use of available BRAMs on the FPGA. A hybrid solution between this shared memory and the local memory would be sharing a single or

[1] Having five image lines in scope allows the fabric to run frame-based algorithms using 5 by 5 filters.

[2] Double buffering divides the local memory into two buffers. One of the buffers contains a complete frame, that a processor can read from and operate on. The other buffer is used by the processor to build the new frame in. Whenever a new frame is completed, the buffers change their role.

double port memory with multiple cores. In this scenario, a policy regarding accessing the memory is needed, preventing the cores from creating hazardous situations in the memory that occur whenever multiple cores access the same location in a memory at the same time. This inevitably introduces stalls on memory accesses, which increases the latency of the targeted design even more.

4.5.3 Caches

As described in Section 2.4.2, data and instruction caches are not essential components in a streaming multiprocessor, and should preferably not be implemented in the design. The individual cores do not need any sort of data memory other than their internal register file. The data memory is not needed by the processor at all, since no intermediate results are likely to be stored. An instruction memory is needed, and should be at least 6 KiB in size, as determined in Section 3.4.2.

4.6 Processor pipeline

Now that the design of a single ρ-VEX processor and the optimal memory hierarchy have been determined, the optimal architecture for a streaming multicore processor pipeline design will be described. This section will discuss how the the ρ-VEX cores should be connected to each other, and how they should be enabled to interface with peripherals.

4.6.1 Multicore setup

As described in Section 4.5.2, each ρ-VEX processor has its own local memory and is able to interface with the local memory of a neighboring core. To connect multiple ρ-VEX cores to each other in a way that allows streaming of data, a non-circular daisy chain should be created. The length of this chain is determined by the minimal number of desired steps in the image processing algorithm workload introduced in 3.4. Based on the created algorithm workload, the chain should consist of four processors and local memories. It is important to note that the latency of a design such as this pipeline is limited by the stage that performs the most compute intensive stage. This brings about the need of load balancing in order to optimize the processing speed of the pipeline.

4.6.2 Peripherals

The ρ-VEX processor itself has no means to directly interact with resources outside of the FPGA. To add features that allow this, the open-standard on-chip AMBA AHB bus by ARM, a Memory Interface Generator (MIG) by Xilinx [32] and the IP library GRLIB by Cobham Gaisler AB [33] are used. GRLIB contains a wealth of components that can be implemented to the design, and is designed to work with the AHB bus. Xilinx' MIG enables communication with the FPGA board's off-chip Dynamic Random Access Memory (DRAM), which is needed for debugging and storage of possible intermediate results. Used components from GRLIB include a demuxer, that is used for debug purposes, and the dual port RAM block that is used to implement the local memories introduced in Section 4.5.1. Obviously these additional features are useful and necessary for the design

at this point, but they will impact the resource utilization and operating frequency of the computation fabric negatively. Data paths through the generated memory interface will expectedly form the critical path of the fabric.

4.6.3 Data handling

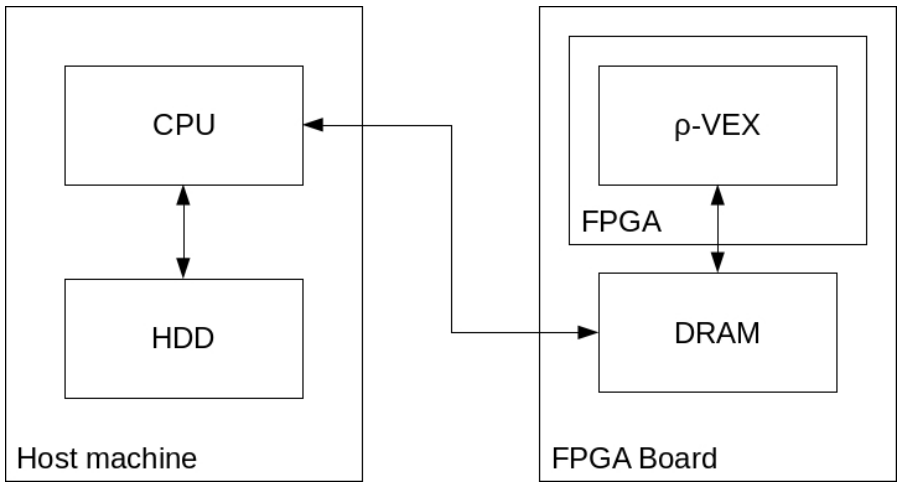
Ideally, the computation fabric should not only be able to stream data through the processor cores, but also between the FPGA board and the host machine. A technique that allows this type of data handling is called Direct Memory Access (DMA). Whenever certain data is needed, a dedicated DMA controller on the FPGA can load this data from an external memory source, like the hard drive of the host machine, directly. Without this technique either the ρ-VEX or the CPU of the host machine would have to process the data transfer. When operating on large sets of data in real-time, as is the case in our medical imaging use case, this type of data transfer would cripple the system by continuously claiming one of the processors. The DMA is able to fetch data from the hard drive on its own, while the host's CPU and the ρ-VEX can continue to work. In this way the latency of transferring data is effectively hidden. When the DMA controller is implemented, interfacing with the off-chip DRAM is no longer needed. This means that generated memory interface can be removed, removing its negative influence on the critical path. This allows the fabric to operate at an even higher frequency. Figure 4.1 shows schematic representations of the data flow in two implementations. Figure 4.1a is the implementation without DMA, where the Central Processing Unit (CPU) of the host machine has to handle data transfers from the host machine's hard disk to the DRAM of the computation fabric. Figure 4.1b shows a configuration that uses a DMA controller. It is clear to see that in such a configuration no processor is involved in transferring data between the host machine and the FPGA board.

4.6.4 Complete design

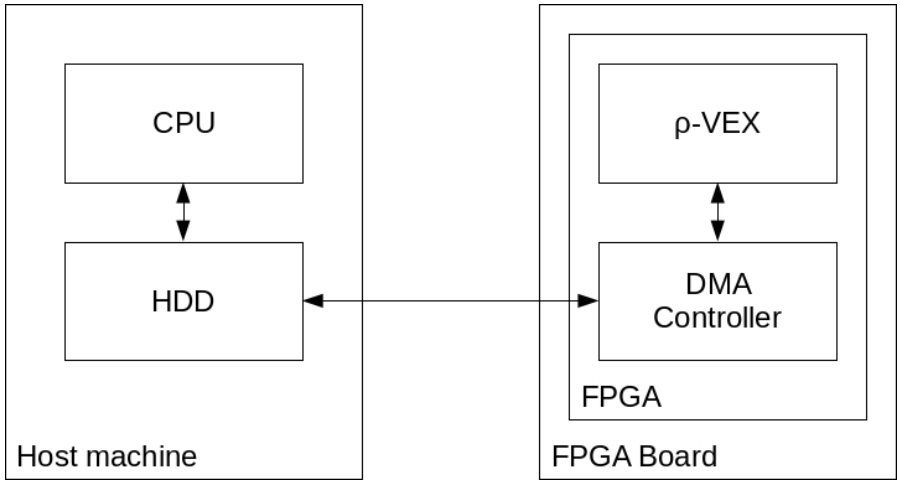
All the choices from this chapter lead up to a final design for our computation fabric. The design consists of a processor pipeline of four two-issue ρ-VEX processors, each of which has been expanded with a local memory for data streaming. The number of processors in the processor pipeline is generic, and can be changed at design-time. All processors of the processor pipeline are connected to an Advanced High-performance Bus (AHB) bus. The computation fabric is handed it's image data by a DMA controller that can interface with the host machine. A schematic representation of the design is shown in Figure 4.2.

4.7 Multi-fabric design

The design of the computation fabric as presented in the previous chapter exploits Instruction-level Parallelism (ILP) and pipelining as forms of parallelism. ILP is supported due to the nature of the ρ-VEX processor, which is embedded in the computation fabric. Pipelining is supported because of the way the ρ-VEX processors are connected within the computation fabric.



(a) Dataflow for a design without a DMA controller



(b) Dataflow for a design with a DMA controller

Figure 4.1: Schematic representations dataflow in two possible designs

In order to also support Data-level Parallelism (DLP), multiple instances of our computation fabric should implemented on an FPGA. In such a structure, a number of lines can be assigned to each instance. Since these instances should all perform the same operations on the data that is provided to them, they can be controlled by a single instruction handler. In doing so, DLP is exploited. Figure Figure 4.3 shows a representation of such a design, where a single instruction memory targets multiple instances of the fabric, which receive their data from the host machine, through a DMA controller.

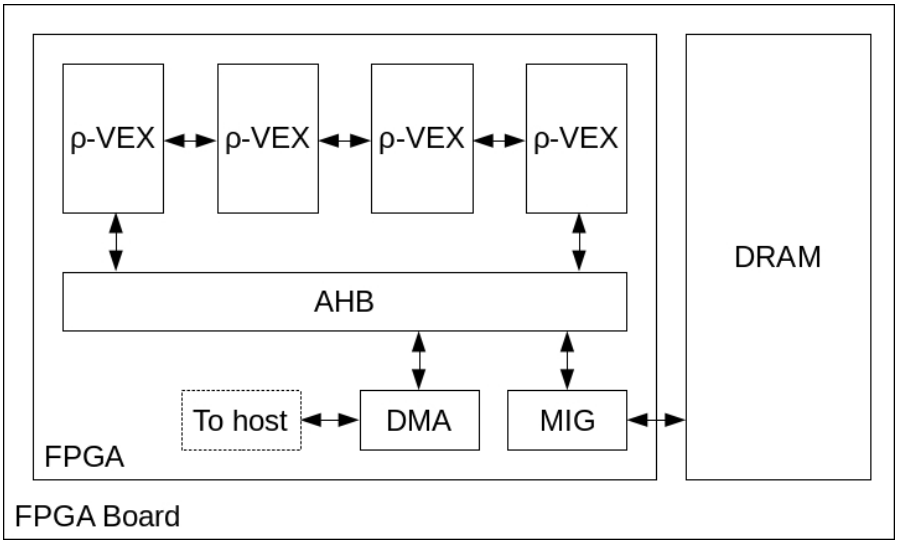


Figure 4.2: The final design for the computation fabric

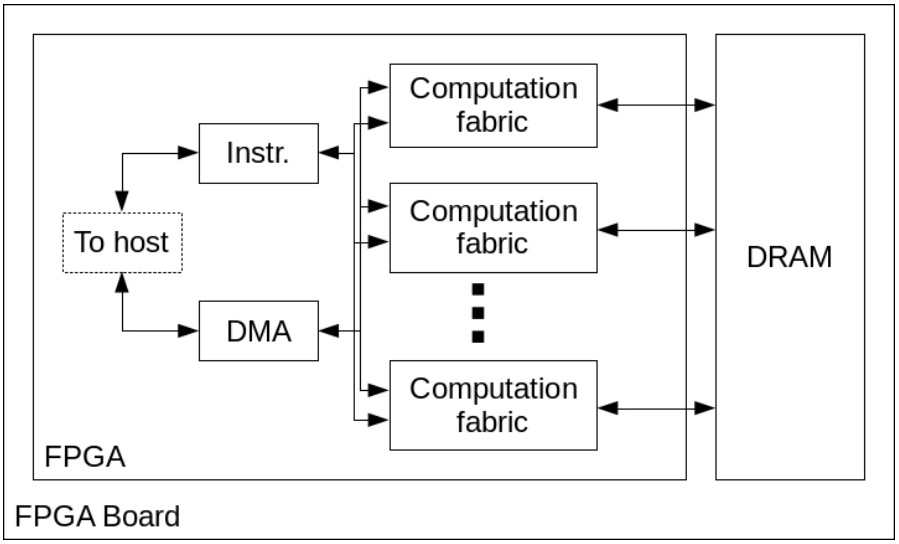


Figure 4.3: The final design for a system with multiple instances of the computation fabric

4.8 Simulations

In order to be able to assess the effect of a local memory based streaming memory hierarchy in a ρ-VEX based system, a simulation tool called *Simrvex* is used [34]. This tool is an adaptation of *xSTSim*, a more generic simulation tool for processors from the ST200 family [35]. Both tools can be made available for academic use. This custom made tool allows the user to easily simulate ρ-VEX based designs, along with peripherals like scratchpad memories. The system described in Section 4.6.4 was implemented in the simulation tool. The system was realized using four ρ-VEX cores, four local memories

and a framebuffer to provide visual feedback of the designs working. Each core is able to read from and write to the local memory that is closest, and read from the local memory that is closest to the previous core. The first core will read the data from source (i.e. the main memory), operate on the data and store it in its local memory. From there the next core can read and process the data, and so on. The image processing algorithms workload construction in Section 3.4 was used for the simulation.

The most important feature of the design, that needs verification by simulation the most, is the streaming memory hierarchy. For this reason, simulations using a streaming and a non-streaming approach were conducted, in order to assess the impact of the streaming memory hierarchy on the design. In the streaming implementation, only the first and the last processor in the processor pipeline interface with the main memory. The data between through the pipeline is being streamed. Each processor has a local memory. In non-streaming implementation, each processor reads the needed data from the main memory of the cache memory of one of the neighboring processors. No data is being streamed. Results of these simulations can be found in Table 4.2. The simulations indicate a speedup of approximately 1.3 times, whenever streaming of data through the computation fabric is enabled. Based on these simulation results, it is safe to assume that implementing the streaming memory hierarchy using small, on-chip local memories is beneficial for the performance of the computation fabric, and effort to implement it will not be in vain.

Table 4.2: Simulation results for streaming and non-streaming implementation of the computation fabric

Problem size	**Clock cycles** (10^6)		**Speedup**
	Streaming	*No streaming*	
240 x 240	1742650	2178813	1.25
480 x 480	6822069	8870680	1.30
960 x 960	27704093	35433015	1.28

5 Implementation

This chapter describes the implementations of our computation fabric based on the design choices described in the previous chapters. A total of two implementations has been created. One fully functional implementation is completed on the ML605 Field Programmable Gate Array (FPGA) board. Another, not fully functional, implementation was created on the VC707 FPGA board in order to be able to indicate maximum achievable results on the available FPGA boards. This chapter will discuss both implementations in order.

5.1 ML605 implementation

The implementation on this FPGA board is completely functional, and is used to acquire all measurement results shown in Chapter 6. This section describes how the design is implemented on this FPGA board.

5.1.1 Processor architecture

The first phase of implementation is getting rid of unnecessary components of the ρ-VEX processor, and making it as small as possible. The second phase is adding the needed functionalities that are not yet present to the processor.

5.1.1.1 Minimizing

The configuration files of the ρ-VEX allow various changes in the system's architecture without having to change any of the source code. By editing the configuration files the following changes have been introduced to the design. The issue width of the core has been reduced to two. Due to this the ρ-VEX processor now exists out of a single core with an issue width of two. Forwarding, traps and tracing have been disabled. The performance counters of the core have been maintained in order to track its performance, but those of the caches have been disabled. The caches could not be removed entirely, since they were too closely connected to the processor's core. Removing them would result in complex issues concerning the data and control flow in the processor. However, the cache sizes have been reduced to a minimum. The data cache size has been reduced to a minimum of 1 KiB. The instruction cache is implemented at different sizes of 2, 4, 8 and 16 KiB in order to asses the impact on the resource utilization of the design. Obviously it is important to ensure that the entire algorithm workload can be stored in the instruction cache for optimal performance. However, assessing the impact of varying cache sizes on the resource utilization of the fabric might be useful to indicate its applicability when working on algorithm workloads of other sizes. As the profiling in

Section 3.4.2 indicates, our algorithm workload has a size of approximately 6 KiB, with taking overhead into account. For this reason, the cores in our implementation will at least need instruction caches of 8 KiB.

5.1.1.2 Rebuilding

With all unnecessary components and properties of the ρ-VEX removed, new features can be introduced. The first of these is realized by another change in the configuration files; the introduction of multiplication units to the ρ-VEX processor core. In most of the algorithms multiplication is a common operation so it is good to equip all cores with dedicated multiplication units. The second change is the introduction of a local memory to each core. As described in Section 4.5.1, this local memory is a fast, dual port RAM block that is used as a scratchpad. One of the ports is connected to the core the local memory belongs to and the other is routed directly to the next core. Sizes of 16, 32 and 64 KiB are used for the local memories, in order to estimate their impact on the design. Larger local memories allow the images to be processed in larger chunks, which reduces the number of needed steps to process a single image. On the other hand, larger local memories do require more resources on the FPGA. Note that for optimal performance of the frame based algorithms from the algorithm workload, a minimal size of 32 KiB is required for each local memory, as is explained in Section 4.5.2.

To ensure correct routing of data transfers an address decoder was added to the processor. Whenever the ρ-VEX core issues a read or write operation, the address is fed to the decoder. Whenever this address is in the range of the local memory associated with the core issuing the operation, the address decoder forwards the address to the local memory. In case of a write, the decoder also forwards the write enable flag, the write mask and the write data. In case of a read, only the read enable flag is sent along with the address. Whenever a read operation is performed on an address that is in the range of the local memory associated with the neighboring core, the address decoder forwards the previously described information to the local memory of the neighboring core along with the stall signal of the core issuing the operation. This signal informs the neighboring local memory how long it should continue to offer the data that is being read from its local memory to the requesting core. Whenever the core that issued the read operation successfully received the required data it will stop stalling and the neighboring local memory can stop offering the data.

5.1.1.3 Final processor

The implementation of the processor as described in the previous section meets almost all the requirements and design specifications formulated in the previous chapter. The only design specification that is not met is the removal of the data cache. A graphical representation of the final processor implementation is shown in Figure 5.1. The processor design is represented in black. The gray parts of the schematic show how the processor could be connected to other processors to form a processor pipeline.

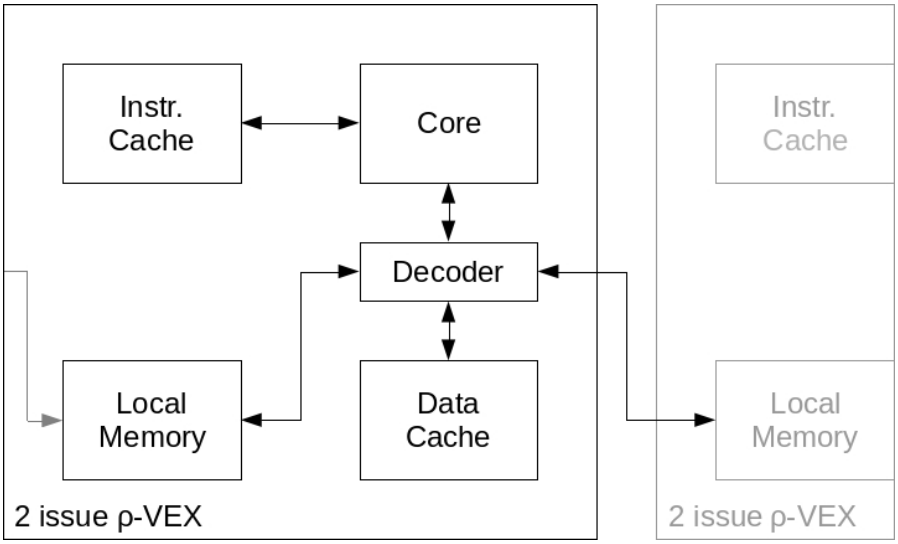


Figure 5.1: Overview of the implemented processor for the computation fabric. The gray shapes show how a processor pipeline can be formed by chaining multiple processors.

5.1.2 Processor pipeline

At this point we have a single two-issue ρ-VEX processor with a single-cycle accessible local memory that can be accessed by the core it's connected to, and a neighboring core. A top level entity was constructed, that connects four of these processors and the peripherals discussed in Section 4.6.2. These peripherals provide essential functionalities to the design, but do have their impact on resource utilization and operating frequency. The number of desired cores in the streaming pipeline is implemented as a generic, allowing the pipeline depth to be altered at design-time. All processors and peripherals are connected to each other over the AHB bus. The local memories of the ρ-VEX processors are connected to each other directly. A graphical representation of this complete system is shown in Figure 5.2. The figure shows that wraparound is supported. This means that the last core in the processor pipeline is able to communicate with the first one. This is a consequence of the implementation of the processor pipeline length as generic. At this time, there is no practical use for the wraparound functionality, but it might possibly be useful and there is no harm in maintaining it for now. Also the figure shows that all cores in the processor pipeline are connected to the Advanced High-performance Bus (AHB) bus, and are thus able to communicate with peripherals if necessary. The image does not show a Direct Memory Access (DMA) controller, since this component could not be implemented. Even though a DMA controller is available in GRLIB, it could not be implemented within the time frame of this thesis project. The fabric uses the FPGA's onboard Dynamic Random Access Memory (DRAM) for storage of images. It uses a memory interface generated by the Memory Interface Generator (MIG) to address the DRAM.

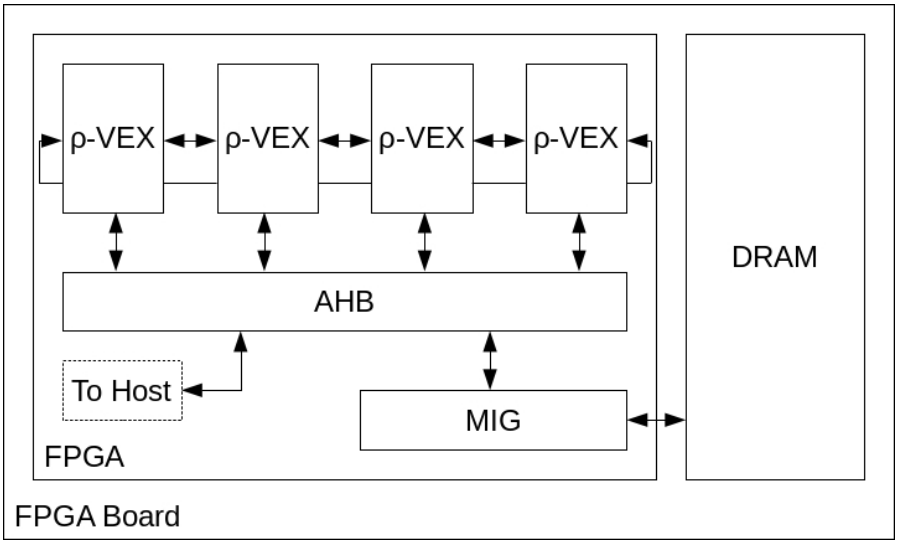


Figure 5.2: Overview of the implemented computation fabric.

5.1.3 Operating frequency

The eight-issue ρ-VEX system suffered from a long critical path, caused by the reconfigurability and internal all-to-all communication possibilities. This limited the operating frequency of the ρ-VEX processor to a maximum of 37.5 MHz. Now that these properties have been removed, the critical paths are drastically shortened and the operation frequency of the system can be increased significantly. Our computation fabric is able to run stable at a frequency of 120 MHz on the ML605 FPGA.

Changing the pipeline in order to achieve an even higher frequency did not succeed at this point. Providing the data memory access stage an additional slot did not bring about the desired results, but corrupted the working of the processor.

5.1.4 Multi-fabric implementation

Only a single instance of the computation fabric has been implemented on the ML605 FPGA board. Results gathered by using this implementation are used to interpolate expected results on an ML605 FPGA board filled with multiple instances of the computation fabric.

5.2 VC707 implementation

The implementation on the VC707 FPGA board is not completely functional, since it lacks both a DMA controller and any form of DRAM on the board. This, and the lack of a good interface with the host machine, make it impossible to run the created algorithm workload on this FPGA. However, several design choices that could not be implemented on the ML605 FPGA board can be implemented on this board. Also, since this board uses newer technology than the ML605 FPGA board, the operating frequency could be

increased further.

5.2.1 Processor architecture

The architecture of the processor in this implementation is roughly the same as that in the ML605 implementation. For this reason, only the differences between the implementation of this processor and that in the ML605 implementation are indicated in this section.

Several unnecessary components of the processor that could not be removed in the ML605 implementation can be removed here. The reason for this is that the ML605 implementation uses a configuration of the ρ-VEX processor that was created to feature GRLIB, and the VC707 implementation uses a standalone implementation of the ρ-VEX processor. A feature of the processor core that could be removed in this implementation is the cache. Both the data and instruction cache were removed. The instruction cache has been replaced by an instruction memory.

Since the local memory was a component from GRLIB, it had to be created by hand. This has been done successfully.

5.2.2 Processor pipeline

Like the ML605 implementation, this implementation also features a pipeline with four cores. However, since it implements the standalone ρ-VEX processor configuration, there are no peripherals. There is no AHB bus, no MIG and no DMA. This means that the critical paths in this implementation are significantly shorter and less resources are consumed. However, due to the lack of these peripherals and the lack of DRAM, this implementation is not functional and can not be used to generate measurement results.

5.2.3 Operating frequency

This implementation is significantly faster than the ML605 implementation, and can operate a frequency of 193 MHz. This is possible due to the lack of a generate memory interface. Also, the instruction pipeline of the processor could be expanded for this implementation without corrupting its working. On top of that, the VC707 features a Virtex-7 FPGA that is superior to the Virtex-6 FPGA of the ML605. This might also be one of the factors that allow this extraordinary high operating frequency.

5.2.4 Multi-fabric implementation

In order to evaluate the maximal achievable performance on the available FPGA boards, the Virtex-7 FPGA has been filled entirely with instances of the computation fabric. A total of 16 instances of the computation fabric could be fitted on the FPGA.

6 Measurements and results

6.1 Experimental setup

This section describes the method used to evaluate various aspects of the developed computation fabric, in terms of the input dataset, the used image processing algorithms, as well as the processor configuration. The used evaluation criteria include the Field Programmable Gate Array (FPGA) resource utilization, operating frequency and performance in frames per second. Also a comparison with the full-size eight-issue ρ-VEX processor is made.

6.1.1 Input dataset

The dimensions of the selected image are 2560 by 1920 pixels, since the ratio of these dimensions works well with our testing environment. The image is resized to other dimensions in order to determine the scalability of system performance. Each pixel is represented by a 32-bit value, as opposed to the 16-bit pixel value used in commercial medical imagers. Using the bilinear interpolation algorithm that is introduced in Section 3.2.1, the image is scaled down to 1280 by 960 and 640 by 480 pixels.

6.1.2 Used algorithm workload

The workload of algorithms based on a typical medical image processing pipeline, as introduced in Section 3.4, is used. The first step in the image processing pipeline is an interpolation algorithm used to scale the size of the source image. Secondly, a gray scaling algorithm is applied. This algorithm is selected because it operates on single pixels in the input dataset. The third stage is a convolution filter that sharpens the image, followed by the final stage, an embossing convolution filter.

6.1.3 Processor configurations

With the use of the described input dataset and algorithm workload, our computation fabric is benchmarked. First of all, we benchmark both architectural configurations: the streaming as well as the non-steaming architectures, and compare their execution times. In addition, due to the importance of optimal memory hierarchy in our streaming architecture, the sizes of local memory and the instruction cache are parameterized to create various processor configurations. This helps in analyzing the design space of the computation fabric and evaluate the impact of the memories on the performance of the processor. All of the configurations implement a streaming-processor pipeline depth of four processors, each of which has a two-issue core running at 120 MHz. Using these

configurations, the optimal trade-off between resource utilization and performance can be found.

Table 6.1: Processor configurations used for benchmarking

Configurations		**Resources**			
LM size (KiB)	*I$ size (KiB)*	*Reg*	*LUT*	*BRAM36*	*BRAM18*
16	2	39894	52035	67	55
16	4	40047	52247	67	55
16	8	40549	52882	67	55
32	2	40074	52973	83	55
32	4	40376	53582	83	55
32	8	40901	53188	83	55
64	2	40260	54466	115	55
64	4	40418	53514	115	55
64	8	40936	54368	115	55

6.1.4 Resource utilization

For this evaluation, we used the Virtex-6 FPGA ML605 Evaluation kit by Xilinx. Specifications of this FPGA board can be found in Section 4.2. Implementing the configurations described in the previous section on this board leads up to the configurations listed in Table 6.1. This table also includes the resource utilization for each configuration. The number of used registers and look-up tables is not significantly impacted by the changing memory sizes. The table also shows that the size of the instruction memory does not impact the Block Random Access Memory (BRAM) utilization for sizes up to 8 KiB. At a size of 16 KiB, the number of needed BRAMs increases. Increasing the size of the local memory has limited influence on the number of BRAMs used, which indicates that we are able to increase the size of the local memory to meet the needs of our application. Note that all BRAMs on the board are 36 KiB in size. So a BRAM36 indicates a fully used BRAM, and a BRAM18 indicates that only half of a BRAM is used.

6.2 Evaluation results

In this section, the results of the performed evaluations are presented. We first discuss the impact of the computation fabric's memory sizes on the computational performance. Based on these results the optimal memory configuration for our application domain is determined. Secondly, we present a comparison between a streaming and a non-streaming variant of this configuration to show the impact of streaming on the fabric's computational performance. Thirdly, we perform a quantitative comparison between our fabric and the baseline ρ-VEX processor. Fourthly and finally, we calculate the expected performance when multiple copies of our fabric are placed on the specific FPGA boards introduced in Section 4.2.

6.2.1 Varying memory sizes

Using the configurations shown in Table 6.1, the timing results listed in Table 6.2 are obtained with the use of the input datasets described in Section 6.1.1. The table shows that varying the size of the local memory has limited impact on performance, reducing execution times by only a few percent. Given the fact that increasing the size of the local memory does not result in any performance gain, a small local memory size should be selected. This ensures a minimal usage of FPGA resources without compromising the fabric's computational performance. However, in order to optimally support medical imaging tasks, the size of the local memory should be at least 32 KiB, as explained in Section 4.5.2. A design with local memories of 64 KiB and instruction caches of 16 KiB was too complex to implement on the FPGA, and hence no results are obtained for this configuration.

Table 6.2: Execution times for configurations with varying memory sizes

Configuration		**Clock cycles** (10^6)		
LM size (KiB)	*I$ size (KiB)*	*2560 x 1920 image size*	*1280 x 960 image size*	*640 x 480 image size*
16	2	2,416.71	612.29	155.31
32	2	2,390.68	603.04	152.43
64	2	2,378.82	597.57	151.47
16	4	732.43	189.67	48.66
32	4	709.95	180.08	45.86
64	4	695.74	175.61	45.17
16	8	398.55	100.32	26.12
32	8	377.21	94.72	23.85
64	8	366.41	92.60	24.24
16	16	393.28	98.23	24.87
32	16	373.04	93.70	23.99
64	16	-	-	-

To give a clear insight in the fabric's computational performance when implementing a 32 KiB local memory, a graph showing the execution times for all configurations with such a memory is shown in Figure 6.1. Note that a logarithmic scale is used to indicate the execution times. Increasing the instruction cache size up to 8 KiB reduces the execution times significantly. Instruction cache sizes of 16 KiB and beyond do not result in further increase in performance. This can be explained by the profiling analysis of the algorithm workload from Section 3.4.2. Whenever the instruction cache is large enough to fit all the instructions of the source code it will no longer experience cache misses, and thus provide maximal performance. We see significant loss of performance when using smaller instruction cache sizes, and no increase in resource utilization when we increase the instruction cache size to 8 KiB.

Based on our findings we select the configuration that features a 32 KiB local memory and an 8 KiB instruction cache as the one that accommodates to our needs in the best way.

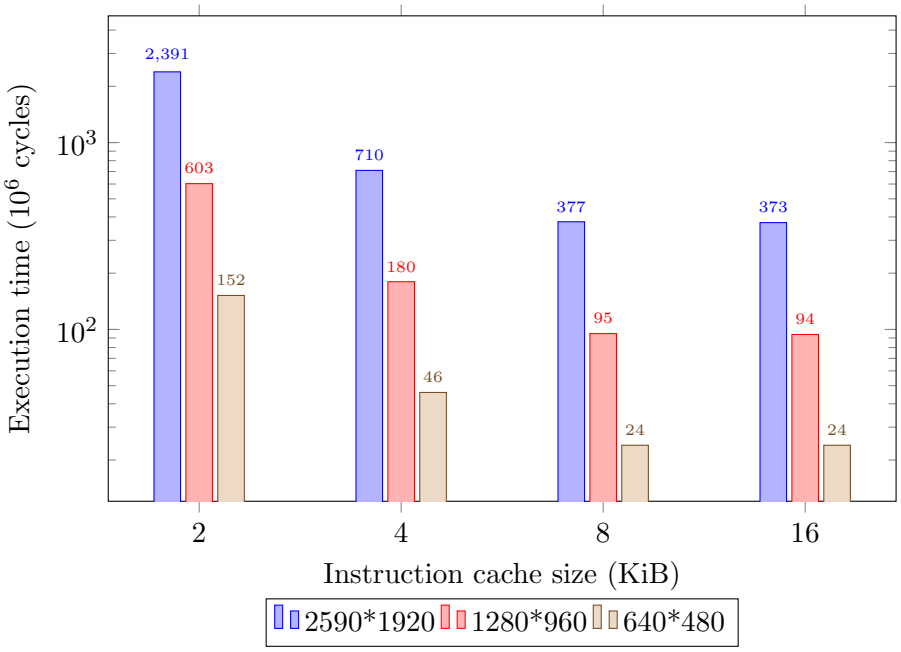


Figure 6.1: Execution times for configurations featuring a 32 KiB local memory

6.2.2 Streaming versus non-streaming

Using the configuration with a local memories of 32 KiB and instruction caches of 8 KiB, the effect of handling data in a streaming fashion is determined. This leads up to the results in Table 6.3 and Table 6.4. The first shows the execution times for bot types of configurations, the second one shows the speedup of the streaming configurations. For reference, measurements for configurations with instructions cache sizes other than 8 KiB have also been included. A graphical representation of the data listed in Table 6.4, is shown in Figure 6.2. Note that a logarithmic scale is used to indicate the execution times. Enabling streaming of data results in speedup of up to 7.59 times. This is significantly higher than the results of the simulations in Section 4.8. The reason for this lies within two factors; memory accesses and bus traffic. Both will be briefly discussed.

Memory accesses Whenever data is being streamed, it has to be fetched and returned just once. Handling data in a traditional, non-streaming way requires the data to be fetched and returned between every processing stage, leading up to four times more memory accesses for our use case. Since accessing the memory is a very time consuming operation, this explains the significantly higher execution times of the non-streaming configuration.

Bus traffic Another factor that increases execution time of this configuration is the bus traffic. The fabric implements four cores that all have a connection to the same bus, that they will all use simultaneously to fetch and return data when operating in a non-streaming way. The bandwidth of the bus is too limited to accommodate the total

amount of data that the four cores try to transport over it, causing the cores to stall until bandwidth becomes available on the bus, thus increasing the execution time even more.

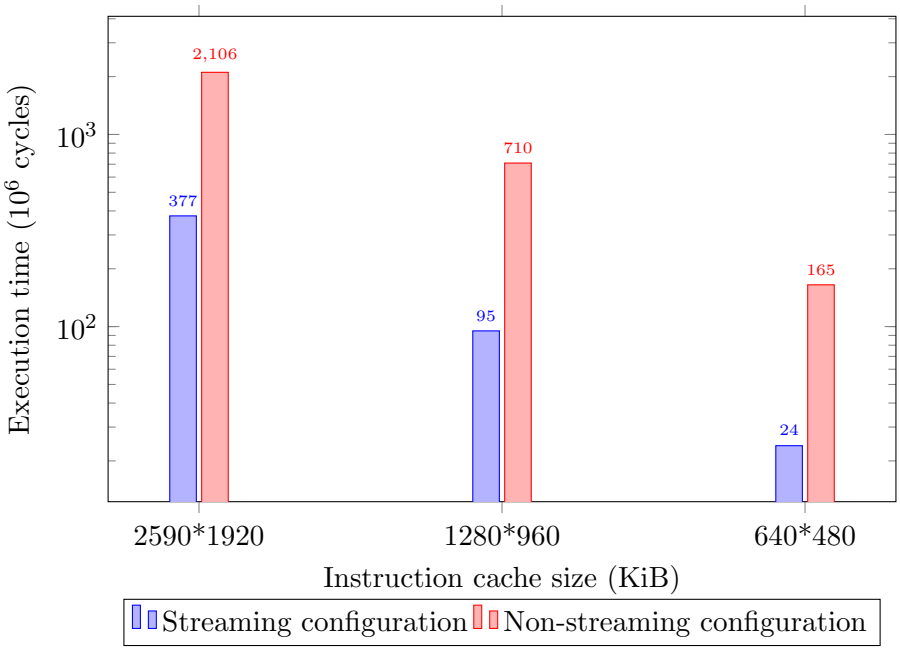


Figure 6.2: Execution times for streaming and non-streaming configurations with 32 KiB local memories and 8 KiB instruction caches.

Table 6.3: Execution times for streaming and non-streaming configurations. The values indicated in red are produced through interpolation, because of limitations in the simulation environment.

Configuration		**Clock cycles** (10^6)		
Streaming	*I$ size (KiB)*	*2560 x 1920 image size*	*1280 x 960 image size*	*640 x 480 image size*
No	2	4,096.59	1,213.15	291.85
Yes	2	2,390.68	603.04	152.43
No	4	2,317.61	782.03	183.11
Yes	4	709.95	180.08	45.86
No	8	2106.36	710.42	165.15
Yes	8	377.21	94.72	23.85
No	16	2109.10	711.43	165.43
Yes	16	373.04	93.70	23.99

Table 6.4: Speedup for streaming and non-streaming configurations.

Configuration	**Speedup**		
I$ size (KiB)	*2560 x 1900 image size*	*1280 x 960 image size*	*640 x 480 image size*
2	1.71	2.01	1.91
4	3.26	4.34	3.99
8	5.58	7.50	6.92
16	5.65	7.59	6.90

6.2.3 Processor architecture considerations

We compare the streaming multiprocessor computation fabric to the standard implementation of the ρ-VEX processor in terms of computational performance, applicability, resource utilization and operating frequency. From this we can deduce how much more powerful our fabric is as compared to the standard ρ-VEX implementation for our selected application domain.

6.2.3.1 Computation performance

The standard ρ-VEX implementation features an eight-issue processor, which is created by combining four two-issue cores. Our fabric features four individual two-issue cores. The only benefit the standard ρ-VEX implementation has over our computation fabric is the ability of operand forwarding in the pipeline of the processor. This technique can prevent pipeline stalls in certain situations. Because of this, the computational performance of the standard ρ-VEX implementation for non-streaming workloads is slightly better when the operating frequency of the implementations is not taken into account. Also in terms of flexibility, our computation fabric outperforms better than the standard ρ-VEX implementation, since our fabric is able to operate in streaming and a non-streaming mode.

6.2.3.2 Resource utilization

Our fabric is smaller than the standard ρ-VEX implementation, as can be seen in Table 6.5. The main reason for this is the absence of the all-to-all crossbar in our fabric, which is a large and costly component of the eight-issue processor. The simple and direct connections between the cores in our fabric come at a very low cost and allow the cores to communicate as well, and even allow streaming.

Table 6.5: Resource utilization comparison standard rVEX and new design

Platform	**Resources**			
	Reg	*LUT*	*BRAM36*	*BRAM18*
Default design	40376	53582	83	55
Default ρ-VEX	54676	84453	57	137

6.2.4 Image processing performance

In order to be able to express the computational performance of our fabric in practical terms, the maximum number of images that can be processed each second is determined. This metric is known as frames per second. First we calculate the achievable number of processed frames per second on the ML605 FPGA board, where a single instance of our fabric is implemented. We estimate the maximum number of times our fabric fits on the Virtex-6 FPGA, and calculate the expected number of processed frames per second on this platform whenever it is filled completely with multiple instances of our computation fabric. Then we repeat these calculations for the Virtex-7 implementation of the VC707 FPGA board, and calculate the maximum achievable number of processed frames per second on the VC707 FPGA board given the achieved implementation results presented in the previous chapter

6.2.5 ML605

Firstly the number of frames that can be processed per second by a single instance of the computation fabric is determined. This result is then used to estimate the maximum achievable number of processed frames per second.

Frames per second per instance A single image, or frame, as used in typical medical imaging products has a size of 960 by 960 pixels. Interpolating the data from Table 6.2 for an image with these dimensions, an performance indication can be given. An image sized 1280 by 960 pixels is 1.33 times larger than a 960 by 960 pixels images. Processing an image sized 1280 by 960 requires 94.72 million clock cycles. From this we can deduce that processing an image sized 960 by 960 pixels would require roughly 71.04 million clock cycles. At an operating frequency of 120 MHz, this would mean that a single instance of our computation fabric can process approximately 1.69 frames per second.

Maximum number of frames per second on Virtex-6 Since our fabric does not only implement two-issue cores, but also a number of peripherals, it is not trivial to determine the resource utilization of a single two-issue core and calculate the maximum number of cores that can be placed on the FPGA board. To be able to determine the resource utilization of a single core, and the resource utilization of the peripherals a number of configurations of our fabric that feature different processor pipeline depths were implemented. Configurations with a depth of 2, 4, 6 and 8 cores were implemented. Their resource utilization can be found in Table 6.6. From this table the resources per core and the overhead caused by the peripherals can be determined. The results of this are listed in Table 6.7. Using these numbers and number of Table 4.1, the maximum number of cores per Virtex-6 FPGA can be calculated. Results of these calculations can be found in Table 6.8. Assuming a four-stage streaming-processor pipeline, the computation fabric would fit four times on a Virtex-6 FPGA. Multiplied by the earlier determined 1.69 frames per second, an efficiently filled Virtex-6 FPGA would be able to process up to roughly $1.69 \cdot 4 = 6.76$ frames per second. A minimum of 15 frames per second is needed to let the human eye interpret the stream of images without significant

difficulty. For optimal interpretation a minimum of 30 frames per second is required. The implementation of our computation fabric on the Virtex-6 FPGA does not meet these requirements, making it not suitable for real-life applications.

Table 6.6: Resource utilization for varying pipeline depths

Pipeline depth	**Resources**			
Cores	*Reg*	*LUT*	*BRAM36*	*BRAM18*
2	33265	36120	55	29
4	40376	53582	83	55
6	46890	69203	111	81
8	53695	85466	139	107

Table 6.7: Resource utilization per core and of peripherals

Resources	**Reg**	**LUT**	**BRAM36**	**BRAM18**
Core	3393	8190	14	13
Peripherals	26592	20143	27	3

Table 6.8: Total number of cores and processor pipelines that can be implemented on the Virtex-6 FPGA

Component	**Maximum amount of possible instances per resource**			
	Reg	*LUT*	*BRAM36*	*BRAM18*
Core	81	16	28	64
Processor Pipeline	20	4	7	16

6.2.6 VC707

Firstly the number of frames that can be processed per second by a single instance of the computation fabric is determined. This result is then used to estimate the maximum achievable number of processed frames per second.

Frames per second per instance As determined in Section 6.2.5, processing of a single image of 960 by 960 pixels requires roughly 71.04 million clock cycles. At an operating frequency of 193 MHz, this would mean that a single instance of our computation fabric can process approximately 2.72 frames per second.

Maximum number of frames per second on Virtex-7 Since a total of 16 instances of our computation fabric can be implemented on a Virtex-7 FPGA, the maximum achievable number of frames per second is expected to be $2.72 \cdot 16 = 43.52$. This number is significantly higher than the 30 frames per second requirement introduced in Section 6.2.5, making the implementation of our computation fabric on the Virtex-7 FPGA applicable for real-time image processing.

Conclusion and recommendations

7

This chapter concludes the work for this thesis project. First a summary of the performed work and achieved results is given. This is followed by an overview of main accomplishments and achievements of this work. The chapter then discusses a list of future work opportunities regarding our computation fabric.

7.1 Conclusions

7.1.1 Summary

In Chapter 1 the problem statement and research questions for this thesis project were posed, along with an overview of prior work related to the work of this thesis. Chapter 2 provided background information on the various types of platforms suited for image processing with their pros and cons, followed by background information on the designs of both the ρ-VEX Very Long Instruction Word (VLIW) processor developed by the Delft University of Technology, and modern Graphics Processing Unit (GPU) architectures. The useful properties and techniques from both of these with regards to the design of our computation fabric are summed up at the end of that chapter. Chapter 3 gives background information on algorithms. General algorithm complexity analysis is explained for both sequential and parallel computation. Image processing algorithm classes are introduced along with examples from each class. A number of these are used to construct an algorithm workload for a typical medical imaging use case. The computational requirements posed by this workload of algorithms are determined. Chapter 4 discusses the design choices for our computation fabric. The memory architecture, core architecture and multiprocessor design overview are the most important parts of the design. Simulation results are included to support the design considerations. Chapter 5 discusses the implementation of our computation fabric. It first discusses the implementation of a minimal size ρ-VEX processor suited for medical imaging purposes. This is followed by a description of the implementation of the complete streaming multiprocessor computation fabric. Chapter 6 presents and evaluates the results of the conducted measurements using our computation fabric and describes both achieved and expected performance results.

7.1.2 Main accomplishments and contributions

In Section 1.2, the two following research questions were introduced.

- Is it possible to design and implement a fixed hardware image processing computation fabric that meets the requirements to be used in commercial grade medical imaging products?

- Can we create a computation fabric general enough to handle changes and updates to the image processing algorithms used in the field?

The computation fabric that is presented in this work is a fast and efficient implementation of a fixed hardware streaming multiprocessor architecture for low latency medical image processing. The computation fabric consists of a processor pipeline that supports streaming through so-called local memories, that are highly optimized for the target application. A total of 16 instances of the computation fabric can be implemented on a modern Virtex-7 Field Programmable Gate Array (FPGA) board and can run at an operating frequency of 193 MHz. This implementation can process up to 43.52 frames per second, which makes this architecture very suitable for real-time image processing. By achieving this, all requirements for the first research question have been fulfilled.

The processor pipeline depth of the computation fabric is generic, and can be changed according to the requirements posed by the algorithm workload. This makes the architecture flexible and general enough to handle changes and updates to the algorithm workload. This fulfills the requirements posed by the second research question.

The results of this work help to advance the academic development of the ρ-VEX processor and its applications. Because of its academic and industrial value, a paper discussing the implementation of our computation fabric was written. This paper was submitted to ARC2017, the international symposium on Applied Reconfigurable Computing. The paper can be found in Appendix A of this work.

7.2 Recommendations for future work

Various functionalities and techniques that might further improve and exploit the capabilities of our computation fabric remain unexplored at this moment. A number of these are discussed in this section.

Instruction set architecture As discussed in Section 4.4, our fabric will likely benefit from an Instruction Set Architecture (ISA) extension. Especially implementing the combined multiply-add instruction is likely to be beneficial for the performance of our computation fabric.

Direct memory access Transferring data between the host and the FPGA board requires processor time of the host processor in the current implementation of our computation fabric. However, it is possible to transfer this data without wasting precious processor time of either the host processor or our computation fabric. A way of achieving this is using a Direct Memory Access (DMA) controller. Such a controller is able to interface with a memory directly, without the intervention of a processor. This means that, under certain conditions, the DMA controller can transfer data while both the host machine and our computation fabric can continue performing their work. The most important condition for DMA to work is the guarantee of hazard-free working condition. This means that while the DMA controller is interfacing with a certain part of the memory, no other component in the entire system is allowed to address this specific part of the memory. In order for our fabric to operate on a stream of input data, a DMA

controller is essential. Without it, the continuous stream of images would simply cripple the performance of the host processor, rendering the entire system inoperable.

Lockstep execution Theoretically it is possible to let all the cores in the streaming multiprocessor operate in lockstep. Lockstep is an operation mode that enforces all cores to run synchronously. In most cases lockstep execution is used for fault tolerance, but it might also be suited for parallel processing. By enabling lockstep execution, the need for memories in between the individual cores of the streaming multiprocessor is removed, since all the cores will hand their data to the next core in the processor pipeline at the exact same time. Lockstep execution requires a means of continuous synchronization of instructions between the cores, for example by sharing an instruction memory or program counter.

Implementation of Virtex-7 FPGA As explained in Section 5.2, the implementation of the computation fabric was ported to a VC707 FPGA board that features a larger and faster FPGA than the ML605 FPGA that was used for our performance measurements. In this implementation the technically unnecessary caches and Memory Interface Generator (MIG) of the ρ-VEX processor were removed from the implementation, allowing a higher operating frequency and a lower resource utilization. Unfortunately it is not yet possible to run applications on this implementation, since it lacks a proper bus that allows communication with the host machine. To really uncover the potential of the VC707 FPGA board, proper communication with the host should be implemented. However, since the VC707 FPGA board does not feature any form of main memory, this implementation step should preferably be accompanied by the introduction of a DMA controller to the design.

Pipeline exploration The operating frequency of our fabric is much higher than that of the standard ρ-VEX implementation. However, even higher operating frequencies might be achieved by exploring the limiting stages in the processor instruction pipeline and granting this stage one or more additional time slots. Explorations on the VC707 FPGA board have already shown possible frequencies of up to 200 MHz.

Implementation in Spark The last suggestion for future work is one that lies outside the design and implementation of the fabric itself. It is a suggestion to implement our computation fabric into a Spark environment. Spark is a cluster based computing framework developed and maintained by Apache, that enables massive parallelization. Since it is open source, our fabric can be implemented as a component in a custom-made Spark environment. A team of researchers from the Computer Engineering group of the Delft University of Technology has obtained remarkable successes with computation platforms in a Spark environment, that might be even improved further when our fabric is added to this specific computing framework.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] NVIDIA Corporation. (1999) NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256. [Online]. Available: http://www.nvidia.com/object/IO_20020111_5424.html

[3] Almarvi. (2014) Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing. [Online]. Available: http://www.almarvi.eu/

[4] S. Wong, T. van As, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor," in *International Conference on Field-Programmable Technology (ICFPT)*, December 2008.

[5] S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor," in *Proc. 17th International Conference on Advanced Computing and Communications*, Bangalore, India, December 2009, pp. 244–251.

[6] J. Hoozemans, S. Wong, and Z. Al-Ars, "Using vliw softcore processors for image processing applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 2015, pp. 315–318.

[7] Computer Engineering Laboratory, EEMCS Delft University of Technology. (2016) -VEX Project Site -vex the dynamically reconfigurable vliw processor. [Online]. Available: http://rvex.ewi.tudelft.nl/

[8] D. Stevens, V. Chouliaras, V. Azorin-Peris, J. Zheng, A. Echiadis, and S. Hu, "Biothreads: A novel vliw-based chip multiprocessor for accelerating biomedical image processing applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 6, no. 3, pp. 257–268, June 2012.

[9] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 27–39.

[10] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[11] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.

[12] S. M. Chai, S. Chiricescu, R. Essick, B. Lucas, P. May, K. Moat, J. M. Norris, M. Schuette, and A. Lopez-Lagunas, "Streaming processors for next-generation mobile imaging applications," *IEEE Communications Magazine*, vol. 43, no. 12, pp. 81–89, Dec 2005.

[13] P. Wang and J. McAllister, "Streaming elements for fpga signal and image processing accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 6, pp. 2262–2274, June 2016.

[14] D. Seidner, "Improved low-cost fpga image processor architecture with external line memory," in *2013 IEEE International Conference on Industrial Technology (ICIT)*, Feb 2013, pp. 1128–1133.

[15] M. V. G. Rao, P. R. Kumar, and A. M. Prasad, "Implementation of real time image processing system with fpga and dsp," in *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*, Jan 2016, pp. 1–4.

[16] G. Bieszczad, "Soc-fpga embedded system for real-time thermal image processing," in *2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems*, June 2016, pp. 469–473.

[17] N. Masud, J. Nasir, M. S. Nazir, and M. Aqil, "Fpga based multiprocessor embedded system for real-time image processing," in *2015 15th International Conference on Control, Automation and Systems (ICCAS)*, Oct 2015, pp. 436–438.

[18] S. McBader and P. Lee, "An fpga implementation of a flexible, parallel image processing architecture suitable for embedded vision systems," in *Proceedings International Parallel and Distributed Processing Symposium*, April 2003, pp. 5 pp.–.

[19] S. O. Cisneros, J. R. D., P. M. Villalobos, C. A. T. C., H. Hernndez-Hector, and J. J. R. P., "An image processor for convolution and correlation of binary images implemented in fpga," in *2015 12th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, Oct 2015, pp. 1–5.

[20] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[21] STMicroelectronics. (2012) RM0245 Reference manual st200 run-time architecture. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/reference_manual/fc/2a/dc/5e/37/ed/41/09/CD17521848.pdf/files/CD17521848.pdf/jcr:content/translations/en.CD17521848.pdf

[22] Hewlett-Packard Laboratories. (2009) VEX Toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[23] NVIDIA Corporation. (2016) Parallel Thread Execution ISA application guide. [Online]. Available: http://docs.nvidia.com/cuda/pdf/ptx_isa_5.0.pdf

[24] ——. (2016) CUDA C Programming Guide design guide. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[25] Khronos. (2016) The OpenCL Specification. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf

[26] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*, 1909.

[27] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, Jun. 1986. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.1986.4767851

[28] Xilinx Corporation. (2012) ML605 Hardware user guide. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf

[29] ——. (2016) VC707 Evaluation Board for the Virtex-7 FPGA user guide. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf

[30] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, May 2002, pp. 73–78.

[31] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings of the 1997 European Conference on Design and Test*, ser. EDTC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=787260.787762

[32] Xilinx Corporation. (2010) Memory Interface Solutions user guide. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf

[33] Cobham Gaisler AB. (2016) GRLIB IP Library user's manual. [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf

[34] J. Hoozemans. (2016) Fast functional simulator for the st200 series of processors. [Online]. Available: https://bitbucket.org/joosthooz/sim-rvex

[35] ERA. (2013) Embedded Reconfigurable Architectures downloads. [Online]. Available: http://www.era-project.org/index.php?option=com_content&view=article&id=56&Itemid=60

[36] C. Pham-Quoc, Z. Al-Ars, and K. Bertels, "Heterogeneous hardware accelerator architecture for streaming image processing," in *Proc. International Conference on Advanced Technologies for Communications*, Hochiminh City, Vietnam, October 2013, pp. 374–379.

[37] V. Kritchallo, B. Braithwaite, E. Vermij, K. Bertels, and Z. Al-Ars, "Balancing high-performance parallelization and accuracy in canny edge detector," in *Proc. 29th International Conference on Architecture of Computing Systems*, Nuremberg, Germany, April 2016.

[38] C. Pham-Quoc, Z. Al-Ars, and K. Bertels, "Automated hybrid interconnect design for fpga accelerators using data communication profiling," in *Proc. 28th International Parallel Distributed Processing Symposium Workshops*, Phoenix, USA, May 2014.

[39] H. Du Nguyen, Z. Al-Ars, G. Smaragdos, and C. Strydis, "Accelerating complex brain-model simulations on gpu platforms," in *Proc. 18th Design, Automation Test in Europe Conference*, Grenoble, France, March 2015.

[40] L. Hasan, M. Kentie, and Z. Al-Ars, "Dopa: Gpu-based protein alignment using database and memory access optimizations," *BMC Research Notes*, vol. 4, pp. 1–11, July 2011.

[41] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "Gpu-accelerated bwa-mem genomic mapping algorithm using adaptive load balancing," in *Proc. 29th International Conference on Architecture of Computing Systems*, Nuremberg, Germany, April 2016, pp. 130–142.

[42] S. Ren, K. Bertels, and Z. Al-Ars, "Exploration of alternative gpu implementations of the pair-hmms forward algorithm," in *Proc. 3rd International Workshop on High Performance Computing on Bioinformatics*, Shenzhen, China, December 2016.

[43] 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.

[44] S. Ren, V. Sima, and Z. Al-Ars, "Fpga acceleration of the pair-hmms forward algorithm for dna sequence analysis," in *Proc. International Workshop on High Performance Computing on Bioinformatics*, Washington DC, USA, November 2015.

[45] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the pairhmm forward algorithm," in *Proc. IEEE International Conference on Bioinformatics and Biomedicine*, Shenzhen, China, December 2016, pp. 758–762.

[46] E. Houtgast, V. Sima, G. Marchiori, K. Bertels, and Z. Al-Ars, "Power-efficiency analysis of accelerated bwa-mem implementations on heterogeneous computing platforms," in *Proc. International Conference on Reconfigurable Computing and FPGAs*, Cancun, Mexico, December 2016.

Publication

A

A paper discussing the implementation of our computation fabric was written. This paper was submitted to ARC2017, the international symposium on Applied Reconfigurable Computing. This paper discusses focuses mainly only the implementation of the computation fabric on the VC707 FPGA board.

VLIW-based FPGA Computation Fabric with Streaming Memory Hierarchy for Medical Imaging Applications

Joost Hoozemans, Rolf Heij, Jeroen van Straten, and Zaid Al-Ars

Delft University of Technology
j.j.hoozemans@tudelft.nl, r.w.heij@student.tudelft.nl, j.van.straten-1@tudelft.nl, z.al-ars@tudelft.nl

Abstract. In this paper, we present and evaluate an FPGA acceleration fabric that uses VLIW softcores as processing elements, combined with a memory hierarchy that is designed to stream data between intermediate stages of an image processing pipeline. These pipelines are commonplace in medical applications such as X-ray imagers. By using a streaming memory hierarchy, performance is increased by a factor that depends on the number of stages (7.5× when using 4 consecutive filters). Using a Xilinx VC707 board, we are able to place up to 75 cores. A platform of 64 cores can be routed at 193MHz, achieving real-time performance, while keeping 20% resources available for off-board interfacing.
Our VHDL implementation and associated tools (compiler, simulator, etc.) are available for download for the academic community.

1 Introduction

In contemporary medical imaging platforms, complexity of image processing algorithms is steadily increasing (in order to improve the quality of the output while reducing the exposure of the patients to radiation). Manufacturers of medical imaging devices are starting to evaluate the possibility of using FPGA acceleration to provide the computational resources needed. FPGAs are known to be able to exploit the large amounts of parallelism that is available in image processing workloads. However, current workflows using High-Level Synthesis (HLS) are problematic for the medical application domain, as it impairs programmability (increasing time-to-market) and maintainability. Additionally, some of the image processing algorithms used are rather complex and can yield varying quality of results. Therefore, in this paper, we propose a computation fabric on the FPGA that is optimized for the application domain, in order to provide acceleration without sacrificing programmability. By analyzing the structure of the image

Part of this work has been supported by the ALMARVI European Artemis project nr. 621439.

processing workload type (essentially a pipeline consisting of multiple filters operating on the input in consecutive steps), we have selected a suitable processing element and designed a streaming memory structure between the processors.

The image processing workload targeted in this paper consists of a number of filters that are applied to the input data in sequence. Each filter is a stage in the image processing pipeline. The input stage of a filter is the output of the previous stage - the stages *stream* data to each other. Making sure these transfers are performed as efficiently as possible is crucial to provide high throughput.

The processing element used in this work is based on a VLIW architecture. These type of processors are ubiquitous in areas such as image and signal processing. They are known for their ability to exploit Instruction-Level Parallelism (ILP) while reducing circuit complexity (and subsequently power consumption) compared to their superscalar counterparts. In the medical imaging domain, power consumption is not a main concern, but as image processing workloads can be divided into multiple threads easily, a reduction in area utilization will likely result in an increase in total throughput.

The remainder of this paper is structured as follows: Section 2 discusses related work, Section 3 discusses the implementation details, Section 4 and 5 present the evaluation and results, and Section 6 provides conclusions and future work.

2 Related work

A prior study on using VLIW-based softcores for image processing applications is performed in [1], showing that a VLIW-based architecture has advantages over a scalar architecture such as the MicroBlaze in terms of performance versus resource utilization. In [2], an FPGA-based compute fabric is proposed using the LE-1 softcore (based on the same Instruction Set Architecture - VEX), targeting medical image processing applications. This work focuses solely on offering a highly multi-threaded platform without providing a memory hierarchy that can sustain the needed bandwidth through the pipeline. A related study on accelerating workloads without compromising programmability is [3], with one of the design points being a convolution engine as processing element. A well-known prior effort, and one of the inspirations of this work, uses softcores to provide adequate acceleration while staying targetable by a high level compiler is the Catapult project [4]. The target domain is ranking documents for the Bing search engine. A related effort that aims to accelerate Convolutional Neural Networks is [5]. However, this project did not aim to conserve programmability (only run-time reconfigurability), as the structure of this application does not change enough to require this. In the image processing application domain, [6] provides a comparison of convolution on GPU or FPGA using a Verilog accelerator, [7] and [8] present resource-efficient streaming processing elements, and [9] introduces a toolchain that targets customized softcores.

3 Implementation

The computation fabric developed in this work consists of two facets; the processing elements and the memory hierarchy, as shown in Figure 1. The implementation of both will be discussed in this section. Then, the process of designing a full platform using these components is discussed.

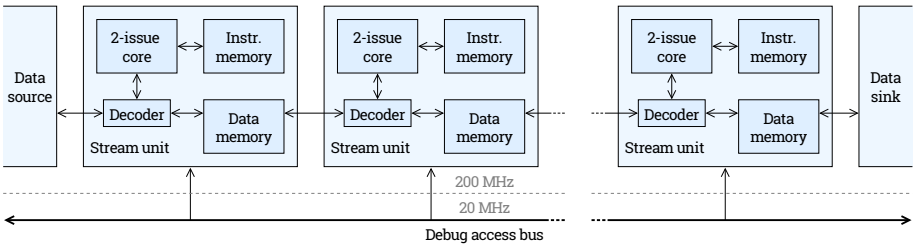


Fig. 1. Organization of a single stream of processing elements (Stream unit) and the streaming connections that link the data memories. Each processor can access the memory of its predecessor. Each processor's memories and control registers can be accessed via a bus that runs on a low clock frequency to prevent it from becoming a timing-critical net.

3.1 Processing elements

This section describes the design and implementation of our fabric. The processor cores in the fabric are derived from the ρ-VEX processor [10]. The ρ-VEX processor is an VLIW processor based on the VEX ISA introduced by Fisher et al [11]. The ρ-VEX processor has both run-time and design-time reconfigurable properties, giving it the flexibility to run a broad selection of applications in an efficient way.

Image processing tasks are highly parallelizable in multiple regards; 1) The code is usually computationally dense, resulting in high ILP, and 2) Every pixel can in theory be calculated individually and it is easy to assign pixels to threads (by dividing the image into blocks). In other words, there is an abundance of Thread-Level Parallelism (TLP). Exploiting TLP is usually more area efficient than exploiting ILP - increasing single-thread performance comes at a high price in power and area utilization and will quickly show diminishing returns. This is why GPUs exploit TLP as much as possible by using many small cores. Therefore, the processing elements of our fabric will use the same approach and we will use the smallest 2-issue VLIW configuration as a basis. This will still allow it to exploit ILP by virtue of having multiple issue slots and a pipelined datapath. By placing multiple instances of our fabric on an FPGA, TLP can be exploited in two dimensions; by processing multiple blocks, lines or pixels (depending on the filter) concurrently, and by assigning each step in the image

processing pipeline to a dedicated core (pipelining on a task level in contrast to the micro-architectural level).

To explore the design space of the processor's pipeline organization, we have measured code size and performance of a 3x3 convolution filter implemented in C. This convolution code forms a basis with which many operators can be applied to an image depending on the kernel that is used (blurring, edge detection, sharpening) so it is suitable to represent the application domain. The main loop can be unrolled by the compiler using pragmas. Figure 2 lists the performance using different levels of loop unrolling for different organizations of a 2-issue ρ-VEX pipeline; the default pipeline with 5 stages and forwarding, one with 2 additional pipeline stages to improve timing, and one using the longer pipeline and with Forwarding (FW) disabled to further improve timing and decrease FPGA resource utilization. Loop unrolling will allow the compiler to fill the pipeline latency with instructions from other iterations. The performance loss introduced is reduced from 25% to less than 2% when unrolling 8 times. Additionally, disabling forwarding reduces the resources utilization of a core allowing more instances to be placed on the FPGA (see Figure 3).

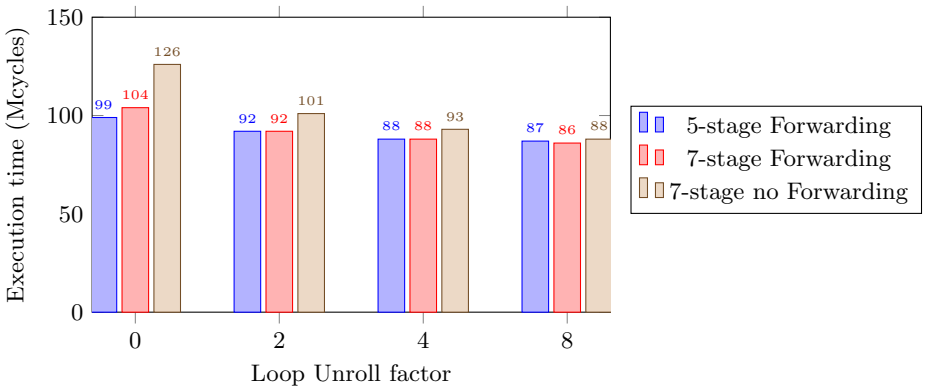


Fig. 2. Execution times of a 4-stage image processing pipeline on a single stream of 4 processors using different loop unrolling factors.

3.2 Memory hierarchy

In our fabric, processing elements are instantiated in 'streams' of configurable length. This length should ideally be equal to the number of stages in the image processing pipeline. Each stage will be executed by a processor using the output of the previous processor. A connection is made between each pair of ρ-VEX processors in a stream, so that a core can read the output of the previous step (computed by the previous core in the stream) and write the output into its own data memory (making it available for reading by the next core in the stream).

The memory blocks are implemented using dual-port RAM Blocks on the FPGA. Each port can sustain a bandwidth of one 32-bits word per cycle per port, so both processors connected to a block (current, next) can access a block without causing a stall. The blocks are connected to the processors by means of a simple address decoder between the memory unit and the data memories.

The first and last core should be connected to DMA (Direct Memory Access) units that move data to and from input and output frame buffers (eventually going off-board).

3.3 Platform

The VHDL code of the components is written in a very generic way and there are numerous parameters that can be chosen by the designer. First of all, the ρ-VEX processor can be configured in terms of issue width, pipeline configuration, forwarding, traps, trace unit, debug unit, performance counters, and caches. Secondly, there is an encompassing structure that instantiates processors in streams. The number of streams and length per stream are VHDL generics.

4 Experimental setup

This section describes the method used in the to evaluate various aspects of the developed computation fabric, in terms of the input dataset, the used image processing algorithms, as well as the processor configuration. The used evaluation criteria include the FPGA resource utilization, operating frequency and performance in frames per second. Also a comparison with the baseline ρ-VEX processor is made.

4.1 Input dataset

Since the target application of the designed system is related to medical image processing, an X-ray sample image is used as input for the evaluation. Typical medical imagers work with images that have a size of 1000 by 1000 pixels. The dimensions of our benchmark images are 2560 by 1920 pixels. The image is resized to other dimensions in order to determine the scalability of system performance. Each pixel is represented by a 32-bit value (RGBA). Using a technique described in the following section, the image may be scaled down to 1280 by 960 and 640 by 480 pixels.

4.2 Used algorithms

A workload of algorithms based on a typical medical image processing pipeline is used. The first step in the image processing pipeline is an interpolation algorithm used to scale the size of the source image. The bi-linear and nearest neighbor interpolation algorithms both have the same computational complexity making them equally feasible. Because of its slightly higher flexibility, we

select the bi-linear interpolation algorithm for the evaluation. Secondly, a gray scaling algorithm is applied. This algorithm is selected because it operates on single pixels in the input dataset. The third stage is a convolution filter that sharpens the image, followed by the final stage, an embossing convolution filter.

The block-based filters in this workload (the convolution steps) operate on 3 by 3 pixel windows. In order to efficiently stream these filters, 3 lines of pixels from the input image need to be available in the local memories. For a typical medical image this would come down to $1000 \times 3 = 3000$ pixels. Given that each pixel has a size of 32 bits, this image fraction has a size of nearly 12 kiB. Double buffering is needed in order to allow a processor in a stream to work on his current output block, while keeping his previous output stable for the next processor in the stream to use it as input. Therefore, the minimum size of a local memory should be at least 24 kiB. The closest value to this that can be realized is 32 kiB.

5 Evaluation results

5.1 Resource utilization

We have synthesized the platform using various configurations targeting the Xilinx VC707 evaluation board. As stated, the pipeline organization of the processing elements has influence on the resource utilization and timing. In Figure 3, 4 options have been evaluated using the standard synthesis flow (unconstrained). With forwarding enabled, the platform completely fills the FPGA using 64 cores. When forwarding is disabled, this can be increased to 75.

Additionally, we have performed a number of runs where we created simple placement constraints that steered the tool towards clustering the cores per stream so that they are aligned on the FPGA in accordance with their streaming organization. A single stream consisting of 4 cores achieves an operating frequency of 200MHz. Using 16 streams, timing becomes somewhat more difficult as the FPGA fabric is not homogeneous (some cores will need to traverse sections of the chip that are reserved for clocking, reconfiguration and I/O logic, and the division of RAM Blocks is not completely uniform). Still, this configuration achieves an operating frequency of 193 MHz at 80% LUT utilization, leaving room for interfacing with off-board electronics.

5.2 Streaming vs non-streaming

In this section, the effect of handling data in a streaming fashion is measured. The results are depicted in Figure 4. Enabling streaming of data results in speedup of 7.5 times. Note that the difference will increase with the number of stages, so the fabric will perform better with increase complex image processing pipelines.

Pipeline organization		Cores	Resource utilization			Freq.
Forwarding	Stages		LUT	FF	BRAM	(MHz)
Enabled	7	64	99%	29%	81%	149
Enabled	5	64	93%	26%	81%	103
Disabled	7	75	96%	33%	95%	162
Disabled	5	75	98%	30%	95%	143
Disabled	7	4	5%	2%	5%	200
Disabled	7	64	82%	28%	81%	193

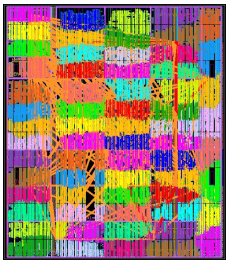


Fig. 3. FPGA Resource Utilization and clock frequency of different platform configurations. The layout of the 64-core, 193MHz platform on the FPGA is depicted on the right.

5.3 Image processing performance

To be able to express the computational performance of our fabric in practical terms, the maximum number of images that can be processed each second is determined.

Processing an image sized 1280 by 960 requires 94.72 million clock cycles (see Figure 4). Using 16 streams At an operating frequency of 193 MHz, this would mean that our fabric can process approximately 34 frames per second.

6 Conclusion

In this paper, we have introduced and evaluated an implementation of a FPGA-based computation fabric that targets medical imaging applications by providing an image processing pipeline-oriented streaming memory hierarchy combined with high-performance VLIW processing elements. We have shown that the

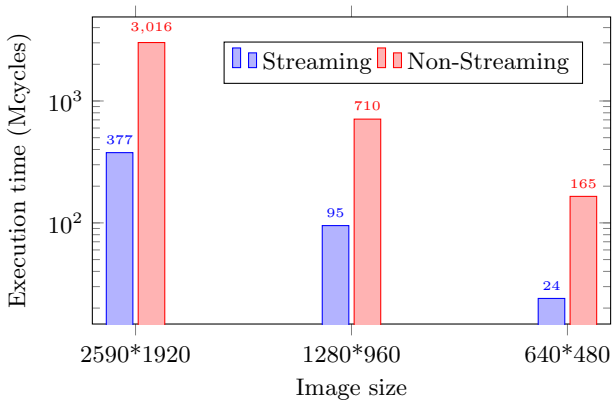


Fig. 4. Execution times of a 4-stage image processing pipeline on a streaming versus non-streaming platform using different image sizes

streaming memory hierarchy is able to reduce bandwidth requirements and increase performance by a factor of 7.5 times when using a single stream of only 4 processing stages. The platform stays fully targetable by a C-compiler and each core can be instructed to perform an individual task. The platform is highly configurable and designers can modify the organization to best match their application structure. For future work, there is room for further design-space exploration of the processing elements in terms of resource utilization versus performance, introducing design-time configurable instruction sets, increasing the clock frequency, and other architectural optimizations. The platform, simulator and toolchain are available for academic use at www.rvex.ewi.tudelft.nl.

References

1. J. Hoozemans, S. Wong, and Z. Al-Ars, "Using VLIW Softcore Processors for Image Processing Applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pp. 315–318, IEEE, 2015.
2. D. Stevens, V. Chouliaras, V. Azorin-Peris, J. Zheng, A. Echiadis, and S. Hu, "BioThreads: a novel VLIW-based chip multiprocessor for accelerating biomedical image processing applications," *IEEE transactions on biomedical circuits and systems*, vol. 6, no. 3, pp. 257–268, 2012.
3. T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 27–39, IEEE, 2016.
4. A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.
5. K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks using Specialized Hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.
6. L. M. Russo, E. C. Pedrino, E. Kato, and V. O. Roda, "Image Convolution Processing: A GPU versus FPGA Comparison," in *2012 VIII Southern Conference on Programmable Logic*, pp. 1–6, March 2012.
7. P. Wang, J. McAllister, and Y. Wu, "Soft-core Stream Processing on FPGA: An FFT Case Study," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2756–2760, May 2013.
8. P. Wang and J. McAllister, "Streaming Elements for FPGA Signal and Image Processing Accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 2262–2274, June 2016.
9. B. Bardak, F. M. Siddiqui, C. Kelly, and R. Woods, "Dataflow toolset for Soft-core Processors on FPGA for Image Processing Applications," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, pp. 1445–1449, Nov 2014.
10. S. Wong and F. Anjam, "The Delft Reconfigurable VLIW Processor," in *Proc. 17th International Conference on Advanced Computing and Communications*, (Bangalore, India), pp. 244–251, December 2009.

11. J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. 500 Sansome Street, Suite 400, San Francisco, CA 94111: Morgan Kaufmann Publishers, 2005.