Hala ρ-VEX: Highly-Programmable Dynamically-Reconfigurable FPGA-based Streaming Platform for Image Processing

Saevar Steinn Hilmarsson

CE-MS-2018-08

Abstract

Image processing is found in many fields and in many domains. Advances in digital image capturing technology allows for faster video rates, of higher quality, than has been seen before and that trend continues. With greater resolution and increased data flow there is also a need for faster and better hardware for image processing. As the trend introduced in Moore's law is slowing down, and possibly reaching saturation in the coming years, there is an ongoing search for new and different solutions in processor architecture. The trend went from single core to multi core and many core and now we are looking into other designs like memory streaming architectures and runtime reconfigurable computers. This thesis designs, implements and evaluates a programming interface for a dynamically-reconfigurable memory-streaming platform for image processing with a focus on programmability, power consumption, reconfigurability and performance. An application programming interface (API) is created to aid with new code development for the platform. The API is a library of functions that are run on an ARM processor and are used to setup, and communicate with, a stream of ρ-VEX soft processors running on a field programmable gate array (FPGA). In this research we look at other state-of-the-art solutions, for comparison and inspiration, that focus on programmability, reconfiguration and performance. The platform is reconfigurable at runtime and experiments show that it takes under 200 ms to completely reconfigure the fabric and initialize a new configuration of ρ-VEX processors. The platform is tested on a Zynq-7000 chip from Xilinx. Comparison is made between streaming architecture and a many core setup using the same amount of ρ-VEX soft processors. The results show a speedup of factor of 2 by using a single processing stream of seven cores compared with seven cores individually running the same algorithm. The result is a working fully-programmable and open-source streaming platform for the image processing domain.

TU Delft Delft University of Technology

CE Lab Computer Engineering Laboratory

Hala ρ-VEX: Highly-Programmable Dynamically-Reconfigurable FPGA-based Streaming Platform for Image Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Saevar S. Hilmarsson
born in SAUDARKROKUR, ICELAND

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

Hala ρ-VEX: Highly-Programmable Dynamically-Reconfigurable FPGA-based Streaming Platform for Image Processing

by Saevar S. Hilmarsson

Abstract

Image processing is found in many fields and in many domains. Advances in digital image capturing technology allows for faster video rates, of higher quality, than has been seen before and that trend continues. With greater resolution and increased data flow there is also a need for faster and better hardware for image processing. As the trend introduced in Moore's law is slowing down, and possibly reaching saturation in the coming years, there is an ongoing search for new and different solutions in processor architecture. The trend went from single core to multi core and many core and now we are looking into other designs like memory streaming architectures and runtime reconfigurable computers. This thesis designs, implements and evaluates a programming interface for a dynamically-reconfigurable memory-streaming platform for image processing with a focus on programmability, power consumption, reconfigurability and performance. An application programming interface (API) is created to aid with new code development for the platform. The API is a library of functions that are run on an ARM processor and are used to setup, and communicate with, a stream of ρ-VEX soft processors running on a field programmable gate array (FPGA). In this research we look at other state-of-the-art solutions, for comparison and inspiration, that focus on programmability, reconfiguration and performance. The platform is reconfigurable at runtime and experiments show that it takes under 200 ms to completely reconfigure the fabric and initialize a new configuration of ρ-VEX processors. The platform is tested on a Zynq-7000 chip from Xilinx. Comparison is made between streaming architecture and a many core setup using the same amount of ρ-VEX soft processors. The results show a speedup of factor of 2 by using a single processing stream of seven cores compared with seven cores individually running the same algorithm. The result is a working fully-programmable and open-source streaming platform for the image processing domain.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-08

Committee Members :

Advisor:	Dr.ir. Z. Al-Ars, QCE, TU Delft
Chairperson:	Dr.ir. Z. Al-Ars, QCE, TU Delft
Member:	Dr.ir. J.S.S.M. Wong, QCE, TU Delft
Member:	Dr.ir. T.G.R.M. van Leuken, CAS, TU Delft

Dedicated to Sigrun, my two sons and my mom Sigga for all their help and support

Contents

List of Figures viii

List of Tables ix

List of Acronyms xii

Acknowledgements xiii

1 Introduction 1
- 1.1 Performance vs programmability . . . 1
- 1.2 Project goal . . . 2
- 1.3 Thesis outline . . . 3

2 Background 5
- 2.1 The ρ-VEX stream architecture . . . 5
- 2.2 FPGA . . . 7
  - 2.2.1 Pynq . . . 7
  - 2.2.2 Zynq . . . 8
- 2.3 Image processing . . . 10
- 2.4 OpenCL . . . 11
- 2.5 Related work . . . 12
  - 2.5.1 Catapult . . . 13
  - 2.5.2 MARC . . . 13
  - 2.5.3 LP-P^{2}IP . . . 14
  - 2.5.4 Halide-HLS . . . 15
  - 2.5.5 Discussion . . . 16

3 Reconfigurable hardware design 17
- 3.1 Hala ρ-VEX platform . . . 17
  - 3.1.1 Programmability . . . 18
  - 3.1.2 Development cycle time . . . 18
  - 3.1.3 Reconfiguration . . . 19
  - 3.1.4 Performance . . . 19
  - 3.1.5 Video streaming interface . . . 19
- 3.2 Challenges . . . 20
- 3.3 Implementation details . . . 20
- 3.4 Portability . . . 23
- 3.5 Limitations . . . 24

4 Platform implementation **25**
4.1 Host implementation . . . 25
4.1.1 Context . . . 26
4.1.2 Platform . . . 26
4.1.3 Device . . . 27
4.1.4 Reconfiguration . . . 27
4.1.5 Program . . . 27
4.1.6 Parameters . . . 28
4.1.7 Execution . . . 28
4.2 Device implementation . . . 29
4.2.1 Map algorithms to the streaming architecture . . . 30
4.2.2 How cores communicate . . . 31
4.2.3 Writing back to host memory . . . 32
4.3 Practical considerations . . . 33

5 Experiments **35**
5.1 Setup and hardware . . . 35
5.2 Results . . . 35
5.2.1 Reconfiguration . . . 35
5.2.2 Performance . . . 36
5.2.3 Power consumption . . . 41
5.2.4 Programmability . . . 41
5.3 Comparison . . . 42
5.4 Discussion . . . 42

6 Conclusion and future work **45**
6.1 Conclusion . . . 45
6.2 Future work . . . 47

Bibliography **53**

A **55**

List of Figures

2.1 The ρ-VEX is reconfigurable at run- and compile time, and can adapt to ILP or TLP depending on the algorithm running [1]. 6
2.2 A simple image processing pipeline where three filters are applied. An input image is grayscaled, then blurred using Gaussian blur and finally a Sobel filter is used to detect edges in the image 6
2.3 Each processing element (PE) utilizes ILP, while the rows exploit DLP and the columns TLP. 6
2.4 Overview of a single processing element, or a stream unit, showing how each processor can access the memory of its predecessor as well as its own memory [2]. 7
2.5 Overview of the Pynq development board [3]. 8
2.6 A simplified model of the Zynq architecture [4]. 8
2.7 Overview of the Zynq processing system [5] 9
2.8 Image processing pipeline with 2 convolution filters. Both every pixel in the image, and each stage can be processed concurrently 11
2.9 The first stage of the Bing ranking pipeline mapped to streaming processors [6]. 13
2.10 This graph shows where MARC fits into modern landscape of computing with regards to programmability vs performance [7]. 14
2.11 Sharpening algorithm mapped to LP-P^{2}IP, the last 4 PEs are in bypass mode [8]. 15
2.12 Overview of how Halide code is compiled. Blue blocks are new, green blocks are unchanged/existing Halide compilation passes [9]. 16

3.1 The `Hala ρ-VEX` compile and run-time flow. 18
3.2 A simple ρ-VEX block design with no support for HDMI. 21
3.3 Implemented hardware layouts of the two available designs shown with 10 ρ-VEX processors. 22
3.4 A simplified block design that contains the ρ-VEX streaming platform and full support for HDMI input and output. 23

4.1 Overview of compile and runtime. Kernels and host code are compiled separately but host is responsible, at runtime, to configure the PL and upload kernels to streams. 26
4.2 One rectangle is moved at a time, from input image to a stream. For convolution filtering, extra padding is needed around the rectangle. 28
4.3 Overview of host ρ-VEX cores can be lined up. 29
4.4 A simple 3-stage image processing pipeline. First the input image is greyscaled, then blurred before a Sobel edge detection is applied and the final image is written to output. 30
4.5 On the host side, one thread is spawned per stream. Here the gray-blur-sobel image processing pipeline has been mapped to two cores and two streams as an example. 30

4.6 Functional flow overview on how DMA transfer is setup and initiated in the last core of every stream. 32

5.1 A theoretical roof line model of the ρ-VEX streaming platform, using 10 cores on a Xilinx Pynq development board. 37
5.2 Throughput of 2 simple algorithms compared with the maximum measured bandwidth. 38
5.3 Results from loop unrolling two image processing kernels tested on two streams consisting of one core each. 39
5.4 Single and dual cores streams compared using two separate image processing kernels. Less is better. 39
5.5 Average execution time of a seven stage image processing pipeline run on: (1) a single stream consisting of seven cores and (2) seven single core streams. . . 40
5.6 Total power consumption of the Zynq, SoC, showing near linear increase in power consumption with every new ρ-VEX core added. 41

List of Tables

2.1 High level comparison of the systems introduced. 16

3.1 Available ρ-VEX processing core setups in the current implementation. The rows are number of available streams and the columns represent how many cores can be in each stream. 22

4.1 Each stage of the image processing pipeline analyzed and divided into processor operations per pixel 31

5.1 Results from measuring average reconfiguration time of the `Hala ρ-VEX` platform. 36

List of Acronyms

API Application Programming Interface

APU Application Processing Unit

ASIC Application-specific Integrated Circuit

BRAM Block Random Access Memory

CLB Configurable Logic Block

CPU Central Processing Unit

DDR Double Data Rate

DLP Data Level Parallelism

DMA Direct Memory Access

DSL Domain Specific Language

EMIO Extended Multiplexed Input/Output

FF Flip-flop.

FFE Free-Form Expressions

FPGA Field Programmable Gate Array

HDL Hardware Description Language

GPU Graphics Processing Unit

GPGPU General-Purpose computing on Graphics Processing Units

ILP Instruction Level Parallelism

IP Intellectual Property

LUT Lookup Table

MISD Multiple Instruction, Single Data

OpenCL Open Computing Language

P^{2}IP Programmable Pipeline Image Processor

PE Processing Elements

PL Programmable Logic

POCL Portable Computing Language

PS Processing System

RAM Random Access Memory

ROM Read Only Memory

TLP Task Level Parallelism

UART Universal Asynchronous Receiver-Transmitter

VHDL VHSIC Hardware Description Language

VLIW Very Long Instruction Word

Acknowledgements

I would like to thank:

Zaid Al-Ars
for being my advisor and giving me the opportunity to work on this great project.

Joost, Jeroen, Jacko and Stephan
for their help with the ρ-VEX processor and the memory streaming platform.

Bjarki, Bjossi, Hrannar and Haji
for proofreading various chapters of this thesis.

Sigrun, Sigga, Sindri and Stormur
for tolerating my long working hours while wrapping up this thesis.

Saevar S. Hilmarsson
Delft, The Netherlands
March 22, 2018

1 Introduction

Image processing is found in many fields and in many domains. To name few example cases where real time image processing is used are: medical imaging, drones, satellites, augmented reality devices, autonomous underwater vehicles (AUVs) and computer vision. Advances in digital image capturing technology allows for faster video rates, of higher quality, than has been seen before and that trend continues. With greater resolution and increased data flow there is also a need for faster and better hardware for image processing.

As the trend introduced in Moore's law is slowing down, and possibly reaching saturation in the coming years, there is an ongoing search for new and different solutions in processor architecture. The trend went from single core to multi core and many core. Now we are looking into other designs like memory streaming architectures and runtime reconfigurable computers. The emphasis used to be mostly on performance but now it is also energy efficiency, size and programmability. New solutions have to be easily programmable for fast development of new code a reuse of older algorithms.

There are many solutions available, and widely adopted for processing real time video streams. Central processing units (CPUs), graphics processing units (GPUs), field programmable gate arrays (FPGAs), digital signal processors (DSPs) and application specific integrated circuits (ASICs) are the most common devices used. They all vary greatly in terms of performance and programmability. From easily programmable devices (CPUs) to high performance chips (ASICs).

1.1 Performance vs programmability

CPUs and GPUs are the most utilized processing devices on the market today. They are both easily programmable and very widely adapted. With languages like the open computing language `OpenCL`, a program can be written once and compiles for a variety of devices form different vendors. This is great in terms of programmability but trails behind more specialized solutions in terms of performance and energy efficiency.

Highly optimized ASICs are on the other end of the performance and programmability spectrum. They can be very fast and energy efficient as they are specially designed solutions for one given problem domain. The development cycle for an ASIC can be long and takes months to design. On top of that is an expensive and time consuming production time. ASICs can not be updated after they have been produced so they become obsolete when the application domain changes or a bug is discovered in the design.

FPGAs are both programmable and reprogrammable, like CPUs and GPUs, so they can be updated in the field and do not get obsolete as easily as ASICs. Though FPGAs are programmable, the development cycle is still fairly long for a highly optimized design. Creating a hardware design on an FPGA is almost as time consuming as designing a new ASIC. It requires a highly specialized knowledge of hardware design, functionality and use of specialized tools

and hardware description languages (HDLs). To make programming for FPGAs more appealing for the general developer a number of high level synthesis (HLS) tools have been developed to create a hardware designs using higher level languages like `C/C++`, `OpenCL`, `Haskel` and others. In [10] the use of `OpenCL` for writing FPGA code proofs to be both faster and more efficient than using HDL. The cost of these solutions, compared to CPU programming for example, are long compile and synthesis cycle times.

The need for performance and programmability is indeed present. There exists a great deal of knowledge, in industry and academia, on image and signal processing using high level languages like `C/C++`, `OpenCL`, `Cuda` etc. Bridging this gap between programmability and performance is an active research field and in Section 2.5 some published solutions in the field will be discussed.

1.2 Project goal

This thesis project is based on the work introduced in [2] which in turn is based on the ρ-VEX [11] processor. It is a proof of concept for a memory streaming hierarchy architecture with medical image processing in mind. There exists an implementation of the platform running a Mandelbrot demo utilizing 64 ρ-VEX processing cores. The Mandelbrot demo runs on a VC707 board from Xilinx. It has an HDMI output to showcase real time results produced by the memory streaming architecture, on an external monitor. The code base for the demo is a coupled application running on a Xilinx's MicroBlaze and the ρ-VEX processing cores. The aim of this project is extract main functionality of the code developed for the MicroBlaze into a library of functions that will serve as the a foundation for writing new applications. Making the ρ-VEX streaming architecture runtime reconfigurable would be an interesting addition. I.e., the total number of streams, and the number of ρ-VEX cores within a stream, could be configurable after a program starts executing. These ideas are summarized in the thesis problem statement:

Is it feasable to use the ρ-VEX memory streaming architecture as a general, programmable, run time reconfigurable image processing platform?

To get a clearer idea of what needs to be accomplished in this thesis work the problem statement is broken down into four separate steps, or goals, that all should be implemented. These steps are:

1. Create a general library in `C` that includes all necessary building block to interface a new application with the ρ-VEX streaming device.

2. Implement a functionality so that the number of streams, and cores within a stream, can be dynamically changed at runtime.

3. Shorten the application development cycle by removing the need to create a new hardware design while a new program is being developed.

4. Create an input output interface for video streaming to properly showcase the platform.

1.3 Thesis outline

The remainder of this thesis report is divided into five separate chapters and their content is as follows:

- Chapter 2. **Background** In the background chapter we will have closer look at the ρ-VEX memory streaming architecture and its state at the beginning of this thesis project. The chapter provides background information on the hardware used, i.e., FPGAs, the Pynq development board and the Zynq chip. The chapter contains a section which looks at image processing on a high level and a follow up section on OpenCL. The chapter is concluded with a related work in Section 2.5 which covers four platforms, or implementations, that are of interest to this thesis project.

- Chapter 3. **Reconfigurable hardware design** A design of the new platform is sketched up on a conceptual level in Section 3.1. The challenges that rise during the hardware design of the platform are covered in Section 3.2. It includes timing, resources, portability and video interfacing. In Section 3.3 the implementation details of the design are listed and why some decisions were made are explained. The chapter is then concluded with a section on designs limitations.

- Chapter 4. **Platform implementation** This chapter covers the main implementation of the ρ-VEX streaming platform. It is explained how application code, written for the platform, is divided between host and device code and how the ARM processors communicates with the ρ-VEX accelerator. Section 4.1 introduces how a host program is setup. What is the methodology behind the host libraries and how to write new code for the host. Section 4.2 covers the device side of the platform where it is explained how to write code for the ρ-VEX processing cores, how they communicate and how output is written back to main memory. The chapter ends with a section on some practical considerations regarding the platform's implementation.

- Chapter 5. **Experiments** In this chapter a number of experiments are described. The platform is tested in regards to reconfiguration time, power consumption, performance and programmability. The main results from the experiments are drawn up in Section 5.2 and compared to other related work in Section 5.3. The chapter is concluded with a discussion regarding the results.

- Chapter 6. **Conclusion and future work** Section 6.1 concludes this thesis report and is followed by Section 6.2 where future work is suggested and includes a number of improvements to the platform on both a hardware and software level.

2 Background

Image processing on field programmable gate arrays (FPGAs) is not a new concept. It has been widely adopted, and in many forms. From highly optimized solutions, using hardware description languages (HDLs), that can take weeks or months to develop, to using high level languages, for instance `OpenCL`, to write code in a matter of days or even hours. In this chapter we will look at previous work, that this thesis project is based on, as well as other scientific publications related to this thesis.

First, there is section about the ρ-VEX memory streaming architecture, which is the foundation of the work performed. Secondly, there is a section about the hardware used, FPGAs and the Pynq board, followed by a section on image processing. Lastly, there is a section about `OpenCL`. In related work we will have a look at four different publications, that are not only relevant to the work presented here, but also serves as an inspiration into what can be achieved. The main topic, in the background section, is on FPGAs as accelerators with focus on: Power consumption, ease of development, reconfiguration and length of development cycle.

2.1 The ρ-VEX stream architecture

The foundation of this thesis project is the ρ-VEX memory streaming architecture, introduced in [2] based on [12]. It uses a ρ-VEX processor [13] as its core building block. The ρ-VEX soft core is a very-long instruction word (VLIW) processor based on the VEX (VLIW EXample) instruction set architecture (ISA) [14]. VLIW processors make use of instruction level parallelism at compile time. The compiler is responsible for hiding pipeline latency by scheduling RISC like operation into bundles (wide words) that are then executed in parallel. The ρ-VEX is a compile- and runtime reconfigurable processor with few possible setups, that are shown in Figure 2.1. In order to fit as many processors as possible onto the FPGA fabric, a smaller version of ρ-VEX was used, with a single 2-way data path and without reconfiguration functionality.

In [2], the streaming memory hierarchy architecture is introduced as a hardware accelerator, for medical image processing. Image processing is highly parallelizable, as in theory, every pixel can be computed independent of other pixels in the image. This is what general image processing units exploit with many core architectures. Image processing pipelines are also split into multiple stages where an output of one stage is the input of the next. Figure 2.2 shows a simple image processing pipeline where an original image enters a grayscaling stage, the intermediate pixels go into a blurring stage, and from there they are processed in a final Sobel edge-detection stage.

The ρ-VEX streaming architecture takes advantage of this feature by proposing a grid like structure of how the cores are lined up in rows and columns. The rows take advantage of data level parallelism (DLP), columns of task level parallelism (TLP) and each 2-way ρ-VEX processor utilizes instruction level parallelism (ILP). This setup is visualized in Figure 2.3. If the number of columns is reduced to one, the streaming device can function as a many core, i.e., a set of independent processors all running in parallel with out any streaming functionality. There

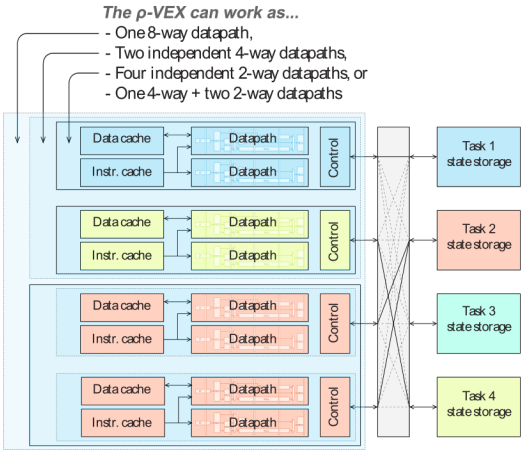


Figure 2.1: The ρ-VEX is reconfigurable at run- and compile time, and can adapt to ILP or TLP depending on the algorithm running [1].

Figure 2.2: A simple image processing pipeline where three filters are applied. An input image is grayscaled, then blurred using Gaussian blur and finally a Sobel filter is used to detect edges in the image

is no global memory present in the design, as is the case with graphics processing units (GPUs), so this is not an ideal setup as the cores can not communicate when working together on a single image.

In Figure 2.4, a single row, or stream, is presented. It shows how a single processor in the stream can write, via decoder, to its own memory or the memory of the next core in the pipeline.

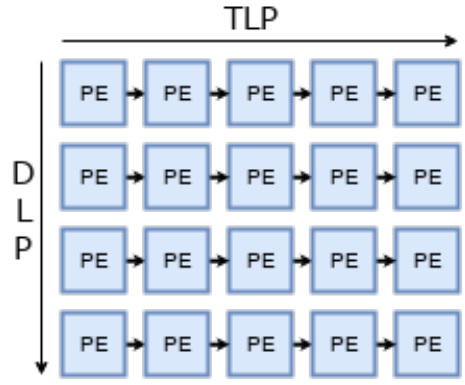


Figure 2.3: Each processing element (PE) utilizes ILP, while the rows exploit DLP and the columns TLP.

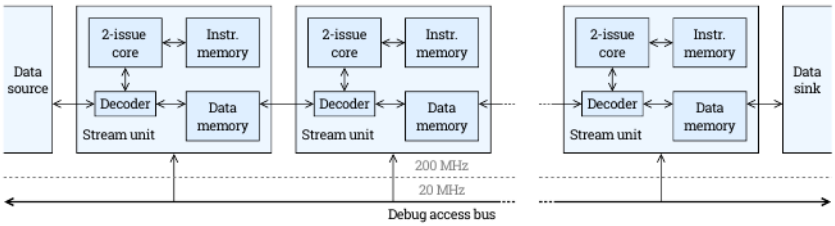


Figure 2.4: Overview of a single processing element, or a stream unit, showing how each processor can access the memory of its predecessor as well as its own memory [2].

This allows for a two dimensional setup that exploits thread level parallelism on one hand, by increased number of streams, and an instruction level parallelism on the other hand, by number of cores per stream.

2.2 FPGA

Field programmable gate arrays (FPGAs) have been around since the 1980s but with recent advancement in FPGA technology and design tools, their popularity have grown significantly. As the name suggests, an FPGA is a two dimensional array of reconfigurable logic gates. This means that a custom hardware can be synthesized and implemented in a far shorter development cycle, than that of application-specific integrated circuits (ASICs), that can take months to develop and manufacture. Also, FPGAs can be updated after being deployed to production, or in the field. Traditionally FPGAs have served as a good tool for rapid hardware prototyping but now with newer and better technology FPGAs have become a viable alternative to ASICs.

FPGAs come in various types and sizes with the two biggest vendors being Xilinx and Intel. In this thesis work a Pynq development board from Xilinx is used. It comes with a Xilinx Zyniq [5] series heterogeneous FPGA-ARM chip. As this is the only board used during this work, further detail on it is given in the following subsection.

2.2.1 Pynq

A Pynq development board, Python on Zynq, shown in Figure 2.5, from Xilinx is used in this project for testing and developing the ρ-VEX streaming platform.

The Pynq board is designed with Python developers in mind. It comes with a base hardware overlay, filled with modules and Python drivers. A developer familiar with Python can access and control modules on the programmable fabric, without any in-depth understanding of hardware design nor FPGA logic. This is a great gateway for developers into the Xilinx' Zynq computing domain. The board also comes with a preinstalled Ubuntu Linux operating system, so all that is needed is basic understanding of Linux and Python and the user can start communicating with the board's peripherals. The many peripherals on the board are the defining reason for why this board was chosen for this project over other boards with bigger and faster FPGAs. The ability to use Python to start communicating with the FGPA, and running some demos, is a great way to get started using the Zynq and is used in this project to control the HDMI interface.

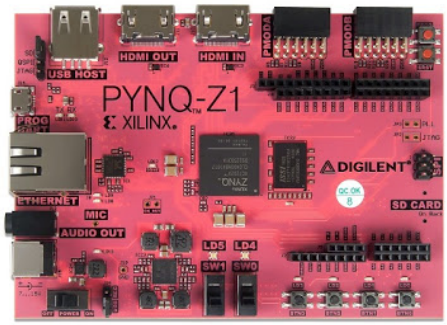


Figure 2.5: Overview of the Pynq development board [3].

In this work, only HDMI-in, HDMI-out and the ethernet ports were used. The central element of the board is a Zynq, all-programmable system on a chip (SoC). The Zynq's defining feature is that it contains both an ARM Cortex-9 dual core processor, and traditional FPGA logic on the same chip. These separate elements, the hard processor, and the programmable logic, are referred to as the processing system (PS) and the programmable logic (PL), respectively. The Pynq board has a 512MB DDR3 memory and a slot for a micro SD card that can be used for memory and storage. As the Zynq is the board's main feature, it will be further detailed in the next subsection.

2.2.2 Zynq

The Zynq chip [5] is a SoC that combines a dual core ARM processor with a traditional reconfigurable logic fabric based on the Xilinx 7-series FPGAs [15]. A full operating system can be run on the ARM processor with a high speed, low latency connection to the programmable logic fabric, via an AXI interface [16]. AXI is a micro controller bus and member of the ARM AMBA family of buses. With the development tools tailored for the Zynq system, the whole design process and implementation cycle becomes relatively short and focused. The chip, on a high level, is split into the PS and the PL, as has been stated and is visualized in Figure 2.6. The PL can be reconfigured at boot-time or by the PS at runtime. It is also possible to only reconfigure the logic fabric partially, leaving some of the logic intact and running. The PL and the PS are further discussed in the next two subsections.

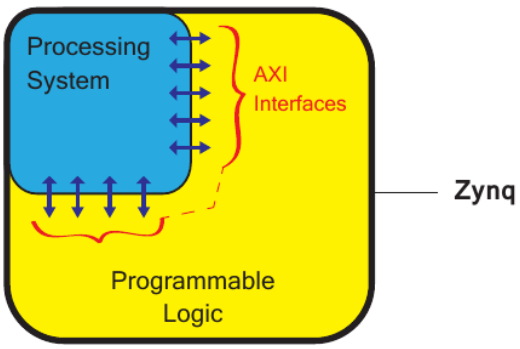


Figure 2.6: A simplified model of the Zynq architecture [4].

Processing system All Zynq devices have an ARM Cortex-A9 [17] processor, as the basis for the processing system [5]. The ARM processor is a hard processor and should not be confused with the soft ρ-VEX processors that are implemented on the reconfigurable fabric. The processing system, Figure 2.7, is not just the ARM but a set of resources that form an application processing unit (APU) as well as clock generators, memory interfaces and interconnect that connects to the PL. The APU has two ARM processing cores, each with a memory management unit (MMU), and level 1 cache. The APU also contains a 512KB level 2 cache and a 256KB on chip memory (OCM) with an OCM AXI4 interconnect that connects to the PL.

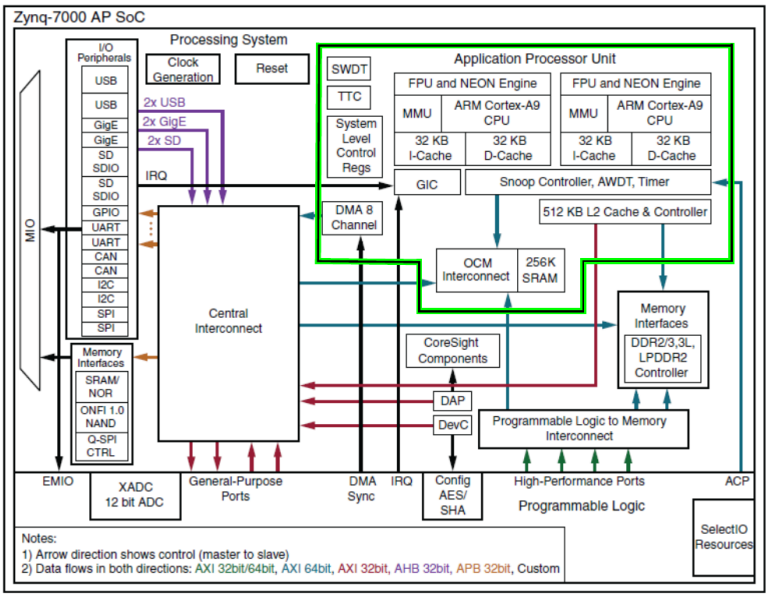


Figure 2.7: Overview of the Zynq processing system [5]

The PS is connected to the PL via an AXI interconnect that serves as a bridge between the two. There are other connection links, like the extended multiplexed input/output (EMIO), but as AXI interconnects were solely used, the EMIO will not be further discussed. The AXI4 standard comes in three flavors, AXI4, AXI4-Lite and AXI4-Stream [16]. The AXI4 protocol provides the highest performance, is memory mapped, and supports burst of up to 256 data words. AXI4-Lite is also memory mapped, but is a simplified version and does not support data burst, a simple use case for the AXI-Lite would be to serve as a configuration bus for modules on the programmable fabric. AXI4-Stream is not memory mapped but supports high speed data streaming with data bursts of unrestricted size. The stream protocol is good for signals that do not need to go to the PS, but can flow through the board.

As the PS can run a full operating system, reconfigure the PL and start data streaming to the logic, it can be seen as a the master where the PL is the slave. This is not always the case, as the PL can run independently of the PS, i.e., be configured and started at boot time, without the PS running any operating system. But that is not the essence of the Zynq design, as it is not just an FPGA but a powerful SoC combining an ARM with an FPGA.

The processing system on the Zynq is a powerful embedded system on its own, and is a great tool to run in tandem with an FPGA on a single chip.

Programmable logic The PL is a two dimensional reconfigurable array of logic gates and wirings. The building blocks of the reconfigurable fabric are many, but maybe most important are the configurable logic blocks (CLB). CLBs are small blocks of logic that are lined up in a two dimensional array in the reconfigurable fabric. Each CLB consists of two slices and every slice contains four lookup tables (LUTs) and eight flip-flops (FF). LUTs are flexible units that can be used to implement different types of logic units such as a function with a maximum of six inputs, small read only memory (ROM), random access memory (RAM) and a shift register. LUTs can also be connected together to create bigger, more complex logic. The FFs are one bit memory elements with a reset function. Next to every CLB is a switch matrix that is used for routing the inner elements of a block, and to connect multiple CLBs together.

There are other and more specialized resource blocks on the PL side, apart from the general fabric already mentioned. These are the block RAMs (BRAMs) for dense memory requirements and the DSP48E1 slices for high speed arithmetic operations. The BRAMs are lined up in the fabric in columns. Each block can store up to 36Kb of data, but can also be split into two 18Kb blocks, on the Pynq board there are 140 BRAM blocks or just under 1.5Mb in total memory capacity. Close to every BRAM column is a DSP48E1 column, as computation and data storage often go hand in hand. The DSP48E1s are flexible units that primarily comprise a pre-adder/subtractor, multiplier and a post-adder/subtractor with four input ports and a single output port.

The PL is configured by loading a bitstream to the fabric. A bitstream is a file that contains information about how all logic should be lined up, and is usually built by a design tool like Xilinx's Vivado. The bitstream can be loaded at boot time, so that the PL is ready when the board starts up. It can also be loaded, via JTAG port, from off-board source like desktop computer. But the interesting part is to use the PS's operating system to load the bitstream at runtime. A library of different bitstreams, hardware designs, can be stored on the device, or in memory, so the PL can be reconfigured at any point. Another feature of interest is partial reconfiguration (PR). It means that a dedicated part of the programmable fabric can be reconfigured without affecting the rest of the PL, and in far shorter time. This can be used to add logic, like soft cores, without stopping or reconfiguring other logic on the board. Let us say, for example, that the chip is a part of embedded system running on battery power, if the power is running low, so in order to conserve energy consumption part of the logic can be removed without halting other processes running at the same time.

Having a highly flexible PL, where almost any hardware design can be realized in a matter of milliseconds on the same chip as a relatively fast general purpose processor is the foundation that makes this thesis project possible.

2.3 Image processing

With faster camera sensors, better lenses and the demand for low energy consumption, image processing is an active field of research. A typical video process today is 1080p image stream at sixty frames per second, and with color correction and other raw processing it can be computationally heavy, like for example, 120 gigaops/sec [18].

A digital image is split into pixels, where each pixel is a few bytes in size, depending on the format. Commonly, image processing algorithms work on individual pixel, independent of other pixels in the image. Image processing pipelines can be very deep, i.e., with many independent

stages and intermediate results. Figure 2.8 shows a very simple two stage pipeline where a 3x3 convolution filter is applied to an input image, and another 3x3 convolution filter is applied to the intermediate image to produce an output image. Every intermediate pixel can be calculated independently with the function *f*. When three lines have been processed the second filter *g* can start working on producing output pixels. This leaves space for exploiting TLP at the same time as DLP. With deeper image processing pipelines, more intermediate filters, or functions, can work concurrently on different stages of the pipeline while still, individually, computing pixels in parallel.

The fastest and most power efficient way to process a video stream is by using specialized ASICs. As an ASIC has no other job but to process a video, in a predefined way, it can be heavily optimized for performance and energy consumption. There are a few downsides to ASIC though. They are expensive in production, with a long design process that can take months to complete. ASICs also cannot be updated after being released to production, and as the name suggests, cannot be reused for different applications.

GPUs are specially built for image processing. They make use of data level parallelism by implementing thousands of small processing cores per processing unit. These devices have big memories, high data bandwidth, and memory hierarchies specialized for fast image processing. GPUs are easily programmable with industry standards like `OpenCL`, maintained by the Khronos Group [19] and CUDA, a powerful platform developed by Nvidia [20], for Nvidia graphics processors. The downside to GPUs is their energy requirements as they are big power consumers, making them not suitable for embedded devices, that rely on battery power or in data centers where power consumption is also a concern.

FPGAs fit in between the ASIC and GPUs. They use far less energy then GPUs, are not as fast as ASICs but are programmable and therefore do not get obsolete as easily as ASICs. Though `OpenCL` has been adapted for FPGAs [21][22], every change made to an `OpenCL` kernel requires a new synthesis process, with specialized design suites tool-chains. By using soft cores like the Xilinx Microblaze [23] or a ρ-VEX , the logic fabric becomes programmable without the need for resynthesis.

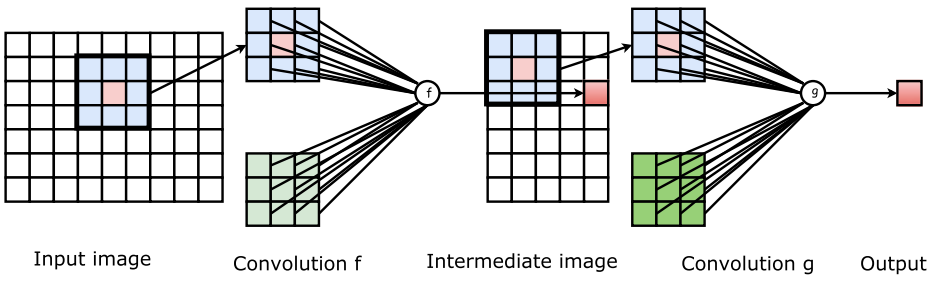


Figure 2.8: Image processing pipeline with 2 convolution filters. Both every pixel in the image, and each stage can be processed concurrently

2.4 OpenCL

`OpenCL` is a widely adopted industry standard for image processing as well as general-purpose computing on graphics processing units (GPGPU). Its strength lies in code portability. In theory,

OpenCL code written for one platform should be runnable on a different one. Code written for an Intel CPU should just as well work on an AMD GPU. Sometimes minor tweaks are needed when code is ported, but in general it should work out of the box. That said, performance does not port as easily, when code is written a developer knows the strengths and weaknesses of the device he is writing the code for. For good performance, it is necessary to understand the memory hierarchy of the accelerator, to know how data is moved and to understand data reuse, like using texture memory on a GPU for example.

OpenCL is an open framework used for writing code for heterogeneous platforms. The application code is divided into host and device code. An OpenCL application is built up in such a way that one host code (runs on CPU) is responsible for setting up devices, downloading kernels and establishing communication links to the device. The code written for an OpenCL device is called a kernel. The kernel is written using a subset of the C programming language and can run on variety of processors, or accelerators, like GPUs, CPUs, FPGAs and DSPs.

The basic steps needed to setup a simple OpenCL host program is to first discover available devices on the host system. A platform finds vendor specific installations, within the operating system, and detects available devices. A context object needs to be created that contains just a single device. After that a program is built, it can be a code source or precompiled binaries ready to run on the specific device. Buffer and queues need to be setup in order to transfer data to the device that has been setup. With this initial setups complete the host can start sending data to the device that executes kernels already uploaded by the program and kernel functions.

One project that is of interest and was going to be the baseline for this thesis work, is a successful attempt to run OpenCL using ρ-VEX as an accelerator [24]. The experiment worked but the results were not great. So an idea arose to use the work already performed and map it to the ρ-VEX streaming architecture discussed in Section 2.1. This was not a one-to-one fit, as it turned out there is no general support for streaming architectures in the OpenCL standard. In order to get the ρ-VEX stream compatible with OpenCL standards, there is a great overhead and it is not given that the work will pay off.

2.5 Related work

Writing programs for FPGAs can be a tedious task that requires specialized tools, in-depth understanding of hardware design, and long development cycles.

It is an interesting task, to make the FPGA more accessible to software developers, while still gaining performance when compared to conventional computing architectures, some solutions have been proposed and implemented.

When developing platforms for image processing on FPGAs there are a few parameters, or Pareto points, that have to be kept in mind. As there is no one solution that works for all, some papers have been written with different results in mind, and in this section, we will look into four solution and compare them on a high level. The problems that need to be addressed are power consumption, programmability, development cycle time, learning curve, streaming capability and performance.

2.5.1 Catapult

In [6] FPGAs are proposed as accelerators in data centers. To maintain data centers homogeneity each server has one FPGA installed. The reconfigurable fabric Catapult is introduced, which is a network of FPGAs in the data center. As a function can be to big to be realized on a single FPGA fabric, the Catapult system can map a function over multiple FPGAs. To test out their new hardware platform, they ported part of the Bing search engine ranking system to the Catapult.

The ranking system of Bing has many long processing pipelines where one data input is processed in more than one way. This feature, multiple-instruction single-data (MISD) computation, means that many streaming computations can be run in parallel. So they propose an array of soft cores, or columns of soft core pipelines working on each data stream as shown in Figure 2.9.

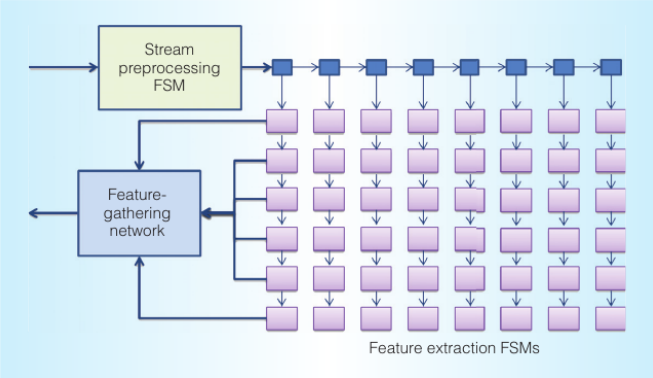


Figure 2.9: The first stage of the Bing ranking pipeline mapped to streaming processors [6].

Rather than using off-the-shelf soft cores a decision was made to develop a new custom soft core processor with focus on massive multithreading and long-latency operations, called free-form expressions (FFEs) processor. By using custom made data paths, composed of FFEs, they managed to increase the ranking throughput by 95% while not increasing overall power consumption by more than 10% [6].

Programmability of the FFEs is not mentioned in [6], but the paper is concluded by stating that programmability is still a major long-term challenge, one can only assume that the Catapult system still requires manual tuning on the hardware design level.

2.5.2 MARC

Many core approach to reconfigurable computing (MARC) is introduced in [7]. The gap, between GPUs and custom made ASICs, seen in Figure 2.10, i.e., the difference between performance and ease of design is attempted to be bridged. FPGAs show great potential in energy efficiency and exploitation of application-specific parallelism. One problem with FPGAs is programmability. The design space, using hardware description languages (HDL), is far away from the experience and expertise of general application developers. MARC proposes a reconfigurable architecture that maintains programmability and performance. Meaning that it should be possible to use the power of FPGAs but write the applications in a high level imperative language like C/C++.

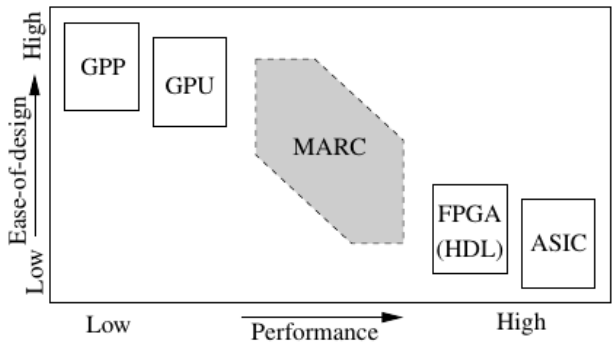


Figure 2.10: This graph shows where MARC fits into modern landscape of computing with regards to programmability vs performance [7].

A many core template is proposed, consisting of a single control processor (C-core) and multiple algorithmic processing cores (A-cores). Every core has a private memory but communication between cores takes place by reading and writing to a global memory shared by all cores.

MARC is compared against manually optimized FPGA implementation of ParaLearn [25], a Bayesian network interface implementation. All experiments were made by synthesizing 48 A-cores and a single C-core. When all C-cores are implemented as RISC processors, MARC only achieves about 5% of the performance shown by the hand tailored solution. To increase the performance, within factor of 3 of the reference, the A-cores had to be application customized during design- and synthesizing phase, taking days to develop.

MARC is a template for a many core system that runs on an FPGA and manages to maintain high level programmability. But when it comes to performance, it still has to be tackled during a hardware design phase, taking days to reach less than half of the performance compared to the highly optimized solution.

2.5.3 LP-P^{2}IP

As power consumption can often be a critical factor, like in drones, satellites, or remote locations, [8] proposes a low power version of a programmable pipeline image processor(P^{2}IP).

The idea introduced is an image processing pipeline with processing elements (PEs) that can be added or removed at runtime by replacing the removed PE with a dummy bypass core. First the total number of PEs has to be decided before the synthesis, as partial reconfiguration (PR) is used to add and remove stages from the processing pipeline. Three basic algorithms were implemented: sharpening, edge detection and corner detection, using three, five and seven PEs, respectively. This means that when a sharpening algorithm is implemented, four PEs can be bypassed and removed resulting in power savings up to 45%.

The whole concept is build on PR. That is, when the hardware is designed, parts of the programmable logic is dedicated to each PE. So that at run time, it is not necessary to stop the image stream and reconfigure the whole logic fabric, but simply switch between bypass cores and PEs, making LP-P^{2}IP a real-time reconfigurable architecture.

As there are seven PEs, and all can be in active or bypass mode, fourteen partial bitstream are stored on an SD card, and at boot time, loaded to DDR memory. They measured the time

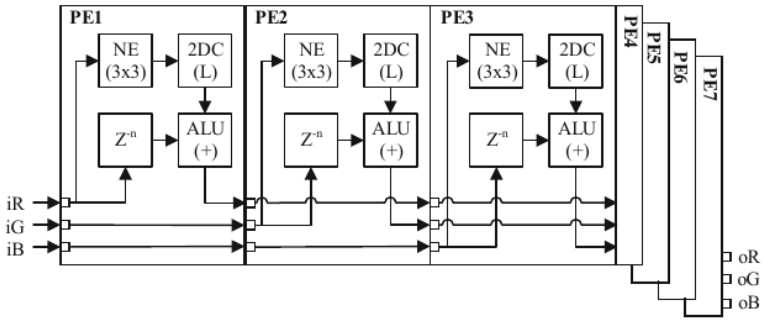


Figure 2.11: Sharpening algorithm mapped to LP-P^{2}IP, the last 4 PEs are in bypass mode [8].

it takes to reconfigure each partial bitstream and the longest is from sharp, three PEs, to corner, seven PEs, or just under twelve milliseconds. It is a considerable speed as full reconfiguration on a Zynq takes just under two hundred milliseconds, so speed up of more than an order of magnitude.

LP-P^{2}IP shows great results in power conservation, up to 45% compared to original implementation, and in runtime reconfiguration. Its down side is that all versions of the image processing pipeline have to be decided at hardware design time, before synthesis, so it can not be changed quickly in the field, or by programmers only familiar to higher level imperative languages.

2.5.4 Halide-HLS

Computational photography, computer vision and augmented reality all require high performance and energy efficiency, as well as ease of programmability and implementation. Writing specialized accelerators on FPGAs is work that needs in depth knowledge of hardware design and using GPUs and CPUs, requires more power. In [9] it is proposed to use a high level, domain specific programming language (DSL), Halide [26]. Halide is popular open source programming language focused primarily on image processing, where algorithm and scheduling is separated to help the user find the most efficient implementation.

Halide-HLS is an open source system, which input is code written in Halide and the output is FPGA accelerators and a Linux source code. The compilation pipeline, show in Figure 2.12, is modified version of the original Halide pipeline.

Six individual image processing application were implemented in [9], which are: Gaussian, Harris, unsharp, stereo, bilateral grid and a camera pipeline. In all cases there were great improvements in performance and energy efficiency. Specially with the Harris corner detection showing 38x and 12x improvement in energy efficiency, and 6x and 3.5x improvement in throughput compared to CPU and GPU, respectively.

The Halide-HLS, proves to be a great tool when it comes to ease of programmability and showed very good results in energy and performance. The downside is that there is no runtime reconfigurability and all changes made to the algorithm or scheduling has to go through the whole compilation pipeline including resynthesizing of the hardware design. I.e, it is not possible to make any changes to an algorithm on the board as an outside work station is needed, with large

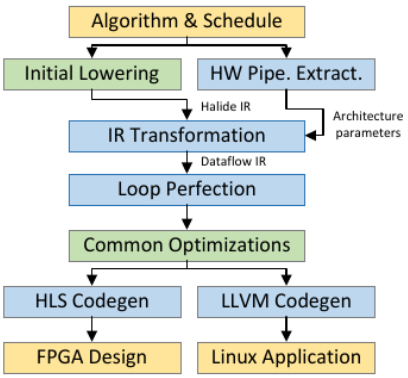


Figure 2.12: Overview of how Halide code is compiled. Blue blocks are new, green blocks are unchanged/existing Halide compilation passes [9].

design tools, to recompile the whole application.

2.5.5 Discussion

In this section we have looked at four different solutions tackling different areas when it comes to ease of programmability, power efficiency, runtime reconfigurability and processor streaming architecture. Now we will try to summarize each system's strong, and weak sides and see if there is a room for yet another solution with focus on streaming architecture and programmability.

Evaluating and comparing the different approaches already discussed, is not easy without implementing them and use predefined benchmarks to test for a quantifiable comparison. This is out of scope of the thesis project. Instead, some high level comparison is made, based on results of published papers, and summarized in Table 2.1

Table 2.1: High level comparison of the systems introduced.

Solution	Programmability	Performance	Development cycle time	Reconfigurable
Catapult	- -	+	- -	-
MARC	-	++	- -	-
LP-P^{2}IP	-	+	-	++
Halide-HLS	++	++	-	-

From the comparison, shown in Table 2.1, Halide-HLS comes out as the best overall solution. With a high level DSL for algorithm and scheduling and a promise of good performance. LP-P^{2}IP also scores well and is the only solution, covered here, that has runtime reconfigurable capabilities. None of the platform score well in all categories so there is a room for a processor streaming solution, with a focus on programmability, power efficiency and run-time reconfiguration.

Reconfigurable hardware design

3

In the previous chapter we had a look at related work and identified a space for a design focusing on processor streaming architecture, programmability, power efficiency and run-time reconfiguration. None of the platforms score well in all categories so there is a room for a new processor streaming solution. It is important to first sketch up a design of the new solution on a conceptual level before moving into implementation details.

In this chapter we will write up a design of our new platform, look back at related work from Section 2.5 and the original problem statement from Section 1.2 and explore what steps need to be taken in order to realize the new platform. Later in the chapter we will move into the platform's hardware implementation starting with challenges in Section 3.2, implementation details in Section 3.3 then this chapter is wrapped up with a section on portability, Section 3.4 and limitations Section 3.5. The platform's implementation will be further explored in the following chapter and evaluated in Section 5.

3.1 Hala ρ-VEX platform

The Icelandic word *halarofa* means moving together in a file, or a row, one behind another. *Halarofa* is a compound consisting of hala, cow's tail, and rofa, dog's tail. In the streaming architecture data flows from one processor to other so the name `Hala ρ-VEX` is a fitting name for the platform.

In this section the `Hala ρ-VEX` platform design is introduced. If we look back at the problem statement, introduced in Section 1.2, the focus is on making the memory streaming architecture [2] programmable and run-time reconfigurable. In the previous chapter we had a look at four different solutions and compared them on a high level. They all scored badly in the *development cycle time* category, only Halide-HLS scored well in *programmability* and LP-P^{2}IP is the only reconfigurable solution that was introduced. It is of interest to create a design that fulfills the steps introduced in the problem statement and fills the gaps presented in related work.

The `Hala ρ-VEX` platform is broken up into five separate design objectives, based on the problem statement and the related work introduced in the background chapter:

1. Programmability
2. Reconfiguration
3. Development cycle time
4. Performance
5. Video streaming interface

In the following subsections we will consider each of these design objectives and develop solutions that will later be implemented.

3.1.1 Programmability

To make the streaming architecture programmable it is important that new code can be written for it using known development patterns or a programming paradigm. The Hala ρ-VEX platform is divided into device on one hand and host on the other. Both host application code and device kernels are written in C but compiled separately, Figure 3.1. In the OpenCL environment, introduced in Section 2.4, a host is a general purpose processor and a device is any OpenCL compatible accelerator or computing device. The design for this platform uses the OpenCL programming model as a reference and it is desirable to make it as OpenCL compatible as possible. That is, we aim at building up libraries so that host code is not too alien for a developer familiar with OpenCL.

An API, used for writing host code, needs to include functions that can find the available ρ-VEX devices, reconfigure the programmable logic, and setup communication lines to the ρ-VEX cores. The API also needs to include ways for a host application to know if the ρ-VEX cores are idle or busy and be capable of uploading new software kernels to the devices.

Code written for the ρ-VEX cores will not be different from writing application for any other ρ-VEX processor. Some factors have to be taken into consideration, like how does one core communicate with another ρ-VEX core and how the streams communicate with the host application. This will be solved by using memory mapped structs that will be initialized within each kernel pointing to the same memory location.

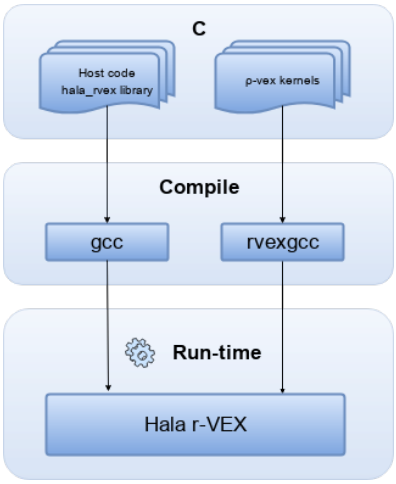


Figure 3.1: The Hala ρ-VEX compile and run-time flow.

3.1.2 Development cycle time

To shorten the development cycle time two things have to be kept in mind: abstract the development process of new application away from the hardware level and make sure that code is standardized to a certain degree. The latter has been covered in the earlier subsection on programmability. To abstract a new application from the hardware it is important that there is no need for a developer writing any hardware description code and that there is also no need for a use of synthesis tools or bitstream generation. To tackle this design criteria all code for the platform is written in C and precompiled overlays are provided.

3.1.3 Reconfiguration

If the platform is reconfigurable that means that if more performance is needed, the number of cores on the logic fabric can be increased. If, on the other hand, energy needs to be conserved, the number of ρ-VEX cores can be reduced at the cost of performance. It is desirable to make the platform runtime reconfigurable, that means it has to happen fast and on the fly.

To be able to call any configuration at run-time, i.e., how many cores there should be on the fabric at any given time, and how they are lined up in terms of stream lengths, it is proposed to prebuild all available setups and store them with the platform's application libraries. This means that when an application asks for, e.g., two streams with 2 cores each, there is a precompiled bitstream that fits that criteria and only needs to be fetched and uploaded at run-time.

3.1.4 Performance

In most image processing applications performance is paramount. With the building blocks, that will make up this platform, there are three parameters that have to specially considered in regards to performance: Clock frequency of the ρ-VEX soft processors has a direct linear impact on performance, number of ρ-VEX processing cores on the fabric and available bandwidth to supply the cores with data. All have their limitations and for different reasons:

- The maximum clock frequency available on the Pynq board's field programmable gate array (FPGA) is 250MHz. As the design gets more complicated it becomes harder to meet timing constrains.
- The total number of ρ-VEX processors that can be implemented on the reconfigurable fabric is restricted by the fabric's size, i.e., available look-up tables (LUTs), memory blocks and arithmetic units. As resources will be used for video interfacing it is an acceptable solution to have up to ten ρ-VEX cores on the fabric as a maximum amount for the platform's implementation on the Pynq board.
- The memory bandwidth is restricted by the board's available interconnect protocols. The fastest way to move data from the processing system over to the programmable logic is via AXI4 interconnect. It can move one data word per clock cycle. The same clock frequency is used for the ρ-VEX soft cores as the AXI4 bus. This means that to exhaust the memory bandwidth a ρ-VEX core would need to produce one result per clock cycle for every new data word. Or in case of ten cores running in parallel, produce one result per core every ten cycles. The arithmetic intensity of image processing pipelines is typically far greater than ten, so using the AXI4 bus should not create a performance bottleneck for the platform.

3.1.5 Video streaming interface

To be able to showcase the platform's performance and usability it is important to interface a live video stream. The Pynq board from Xilinx has both HDMI input and HDMI output. This allows for a web camera to be attached directly to the platform. The boards HDMI input can also be connected to a personal computer as an external monitor. The HDMI output of the board can then be connected to a monitor to showcase its real time video processing capabilities.

The HDMI interfacing is implemented on the programmable logic. Various IP blocks that ship with the Pynq board can be used for this purpose and controlled via Python drivers from the ARM host processor on the Zynq chip. There is a variety of reference designs available on the Internet and one is provided by Xilinx and comes preinstalled on the Pynq board. The reference design that comes with the Pynq board will be used as a reference design for the whole platform's implementation. A problem with that overlay is that it is already packed with logic that uses most of the chip's resources and leaves little space for ρ-VEX processing cores. In order to free up resources, the Pynq overlay will be systematically stripped down leaving only the core HDMI logic left in the design.

3.2 Challenges

During the hardware design, of the `Hala ρ-VEX` platform, several challenges were met. The three biggest ones were interfacing a live video stream, timing the design and resource limitations. I.e., having a working input and output stream of video, meet the designs timing constraints and a utilization of the chip's available resources.

It is possible to fit thirteen ρ-VEX processors on the FPGA with regards to available lookup tables (LUTs) and routing matrices, but with a maximum of 16KB of data memory per ρ-VEX processor. It was desirable to increase the data memory, per processing core, to 32kB at the cost of only fitting 10 on the FPGA as block RAM (BRAM) resources are then exhausted.

There is an example of this particular streaming architecture running on clock speed of just under 200MHz [2] which was set as the baseline, or goal, for this implementation. The Zynq has four clocks that can be fed to the fabric and can be set as high as 250MHz. Still the maximum frequency that was reached was only 85MHz while not violating any timing constraints on the logic fabric.

For a real showcase, one of the main reasons the Pynq board was chosen, it is important to interface the HDMI input and output ports. The Pynq board ships with a base overlay that supports HDMI interfaces. But that same overlay also has many other types of functionality packed in with it using up most of the board's resources. An attempt was made to find another reference design that only implements the HDMI functionality. Several were found, but not with drivers for a Linux operating system as they were all bare metal designs. Following some driver implementation failure it was decided to use the overlay that comes with the Pynq board but systematically remove all extra functionality from the design. The result is an overlay with full HDMI support that uses 17% of the chip's available LUTs and is fully compatible with the Python video drivers available on the board. Using 17% of the LUTs is also an acceptable use of resources as ten ρ-VEX cores, with 32kB memory each, can fit alongside the HDMI interfacing logic.

3.3 Implementation details

The hardware layout is designed on a block diagram level, as there already exists a ρ-VEX streaming block, using Xilinx's Vivado design suite. During development of the reconfigurable streaming ρ-VEX platform, a Pynq-Z1 [3] board from Xilinx was used and all bitstreams generated are for the Zynq-7000 chip. The Pynq has an all programmable system on a chip (SoC)

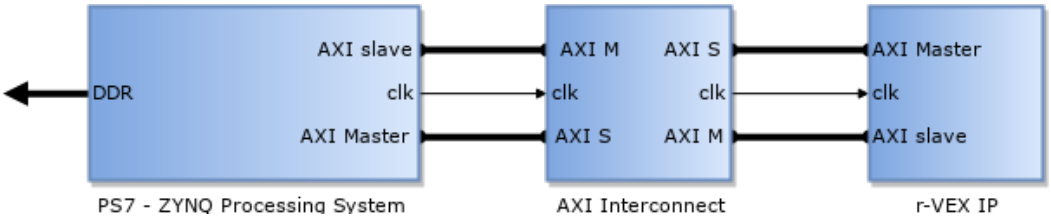


Figure 3.2: A simple ρ-VEX block design with no support for HDMI.

with a dual-core ARM cortex A-9 processor on the same die as an FPGA. The FPGA is based on the Xilinx 7-series architecture family and is on the smaller end of the Xilinx product spectrum. Due to its size a maximum of thirteen ρ-VEX cores can be fitted on the fabric at the same time.

When designing the hardware layout on a block diagram level, there are two main blocks that are of most interest. First it is the ρ-VEX stream intellectual property (IP) block introduced in [2], and second is the Zynq processing system IP. These two building blocks, along with AXI4 interconnects are the bare minimum elements to create a working hardware design that can be run on the Pynq board.

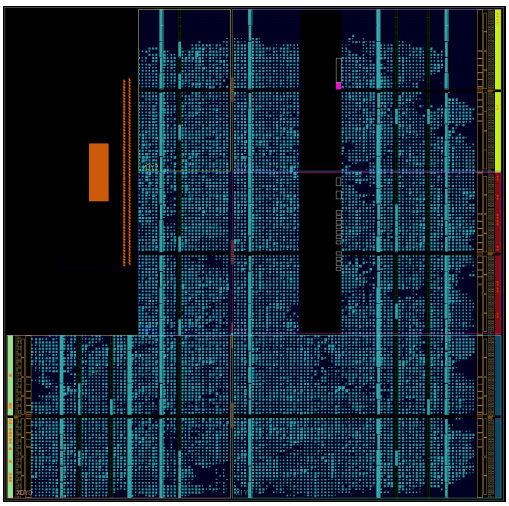
As stated earlier, Xilinx's Vivado is used for designing the hardware layout for the `Hala ρ-VEX` platform. Two basic hardware designs were implemented, one that has support for HDMI input and output, and the other with no extra functionality apart from the streaming ρ-VEX IP block.

The simple design, shown in Figure 3.2, has nothing but a ρ-VEX IP block, an AXI interconnect and a processor system reset. The two external debugging connections, `dbg_tx` and `dbg_rx`, were connected to UART serial port, in the original Mandelbrot implementation, on the VC707 board but are not used in the current design on the Pynq board.

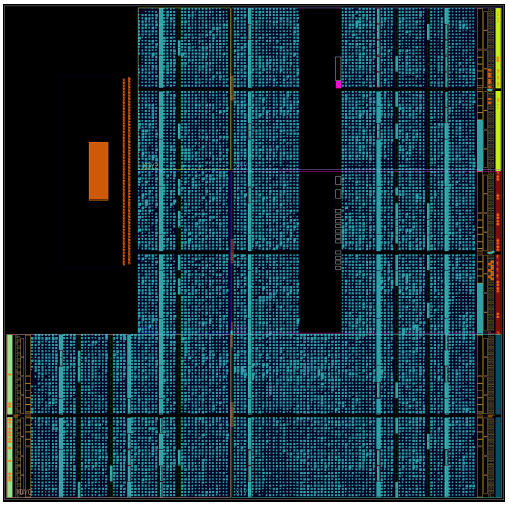
The ρ-VEX streaming IP has to be modified to be compatible with the Zynq-7000 and imported to the Vivado project for the Pynq overlay. Inside the Vivado's block design view, the ρ-VEX streaming module is customizable. By double clicking the ρ-VEX IP within the block diagram view, 13 configurable parameters are available. The ones of interest to this project are:

- Data memory size
- Instruction memory size
- Number of streams
- Number of cores per stream

Other parameters were left at default values in the design process of the `Hala ρ-VEX` platform. It is not possible at this stage to have more then seven cores per stream. This is due to how the cores and streams are memory mapped on the fabric. To take advantage of the full resources of the Pynq board, data memory was increased to 32kB. With the data memory set to 32kB and implementing ten ρ-VEX processing cores on the fabric, all BRAM resources were exhausted and 75% of the available lookup tables (LUTs). The fact that 25% of the LUTs are still available means that three more processors can fit on the PL but with smaller data memories. The actual use of resources is visualized in Figure 3.3a where the use of the PL, with ten ρ-VEX processors, is shown. The figure is generated in Vivado after implementation and optimization.



(a) Hardware implementation of ten ρ-VEX cores on the Pynq board.



(b) Hardware implementation, ten ρ-VEX cores with support for HDMI.

Figure 3.3: Implemented hardware layouts of the two available designs shown with 10 ρ-VEX processors.

Table 3.1: Available ρ-VEX processing core setups in the current implementation. The rows are number of available streams and the columns represent how many cores can be in each stream.

	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓		
3	✓	✓	✓				
4	✓	✓					
5	✓	✓					
6	✓						
7	✓						
8	✓						
9	✓						
10	✓						

The `Hala ρ-VEX` platform is a runtime reconfigurable design. It means that the application developer can make a choice how many processing cores are on the device and how they are lined up. To realize this function of the platform all variations of possible setups are synthesized and implemented. There are 24 different setups available using ten cores as the maximum number on the fabric at once. For example, there can be a single stream consisting of seven cores, or ten streams with a single core each. All the setups available on the Pynq board are shown in Table 3.1.

The second hardware implementation has support for both HDMI input and HDMI output that are connected to the board's physical ports. The HDMI video module, shown in Figure 3.4, is a set of IPs packed together in one block to reduce clutter in the design view. The block contains many IPs needed for the HDMI protocol for pixel conversion. Such as: color convert, pixel pack, DVI to RGB converter, video to AXI stream and more. This hardware design is a stripped down version of the base overlay [27] that comes with the Pynq board. The base overlay

comes packed with functionality out of the box so it is relatively big, or occupied. After searching for HDMI designs made for the Zynq board that come with Linux compatible drivers without much luck it was decided to systematically remove all functionality from the Pynq design except for the HDMI interfacing modules. The results from removing all unnecessary functionality is an overlay with 17% of the LUT resources already utilized. Still ten ρ-VEX cores fit onto the fabric alongside the HDMI block logic.

The actual use of LUTs and BRAM, in the programmable fabric, is visualized in Figure 3.3b alongside the previously introduced design that has no HDMI support. From the two images the difference in resource utilization is visually noticeable.

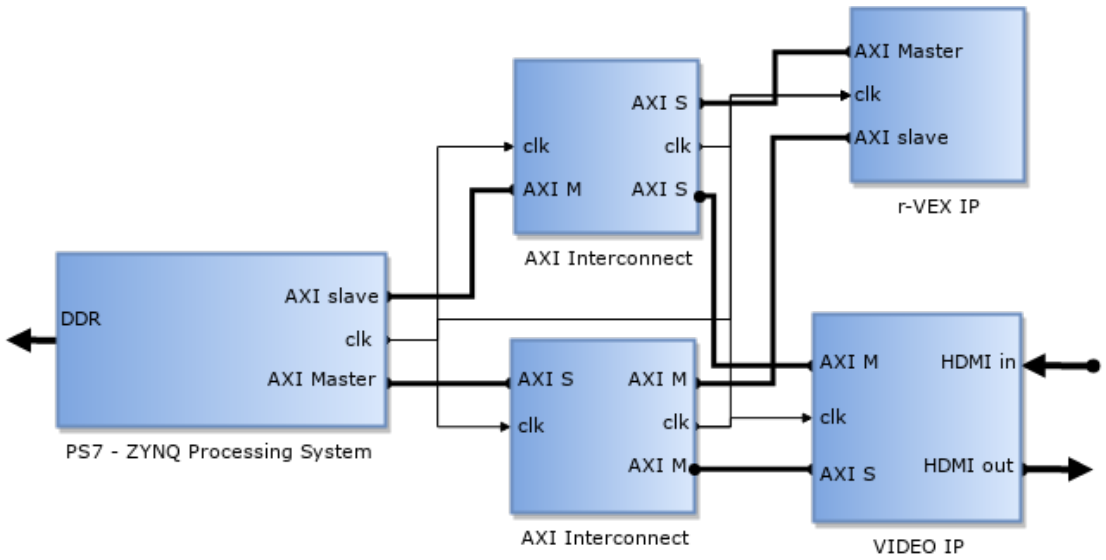


Figure 3.4: A simplified block design that contains the ρ-VEX streaming platform and full support for HDMI input and output.

3.4 Portability

The ρ-VEX memory streaming architecture was previously implemented on a VC707 [28] from Xilinx. It is a lot bigger FPGA and has no ARM on the same chip like the Zynq. That work was ported to the Pynq board, where the ARM is used as a replacement for a Xilinx soft core processor, MicroBlaze, that was responsible for uploading kernels and pushing data to the ρ-VEX streams. The libraries developed during this project have not been tested on the MicroBlaze so it is not known how much work there would be involved porting the project back onto the VC707 board, or any other that has no ARM processor like the Zynq.

As stated already in the beginning of this chapter, the whole work was developed using a Xilinx Zynq chip and the Vivado design suite. To regenerate the project for other Zynq products should be trivial and only a question of the time it takes to regenerate all the bitstreams in Vivado for a targeted chip. That work could be reduced to a script that produces all available configurations given the physical limitations of the target FPGA.

When it comes to other vendors, like Intel, no attempt has been made to realize the platform on their products. The ρ-VEX stream is written in VHDL and the platforms interface in C so in

theory it is portable to other vendors, given some unknown level of modification.

3.5 Limitations

Timing became an issue during the hardware design process. By setting the clock for the ρ-VEX IP block higher than 90MHz the design stopped meeting timing constraints. This was not expected as there exists an implementation [2] where 64 ρ-VEX cores are running concurrently clocking at 192MHz. The design process has to be further explored to increase the clock speed. A possible way to increase the clock frequency is manual routing in order to decrease the critical path, or by putting constraints on how cores are spatially mapped to the fabric. Another solution that was used in a former implementation on the VC707 board was to use custom clock generators and converters. It would be worth exploring similar solutions on the Pynq board. With the clock set so low, it takes eight ρ-VEX cores to match the ARM processor, so increase in clock frequency is desired.

The HDMI in- and output ports make use of the AXI streaming protocol, meaning that data flows through the board without going to memory. There is a video direct memory access (VDMA) block inside the video module. The VDMA, in this particular setup, can write frames to main memory, via the AXI4 protocol, at 568MB/s. In the current implementation there is a lot of data traffic overhead. First the data is converted from an AXI stream and moved to memory via the AXI4 bus, then when ready, the ARM writes the data to the ρ-VEX streams. The last core of the ρ-VEX streaming IP copies data back to main memory and the VDMA fetches the ready frame, converts it into an AXI stream that continues towards the HDMI output. This is a great overhead and a limitation with regards to energy conservation, latency and general use of resources.

The current limitations of the hardware design are seven cores per stream, timing, data movement, available resources and that the design has only been synthesized for a single chip. All five limitations have their separate solutions, like new memory mapping of cores and streams, better routing for improved timing, new logic such as DMA for higher bandwidth, a bigger FPGA and a script that generates bitstreams for different chips respectively.

4 Platform implementation

In this chapter we will explain how the Hala ρ-VEX platform was implemented, how it can be used and go into some discussion on design decisions made in the process. This chapter can also be used as a reference manual for those that want to design new code for the platform. Further information and details on libraries and individual functions can be found in the platform's documentation in Appendix A.

We aimed at making the platform as OpenCL compatible as possible so that existing OpenCL could be modified and compiled for the platform. The emphasize on the OpenCL compatibility was reduced gradually over the time period of this thesis but its heritage is still visible in the platform's design and how it is setup. As was described in Section 2.4, OpenCL host code has to be set up in a predefined order. First, a platform has to be set up, then devices are discovered and a context has to be initialized before setting up queues and buffers. To implement the whole OpenCL standard is out of this project's scope and is not necessarily a good fit as there is no general support for a streaming architecture. Instead a bare minimum set of functions and constants were implemented in a hala_rvex.h with implementations of the functions in a C library.

When writing new applications for the platform it is paramount to understand that though the platform is highly flexible in terms of architecture and number of available cores, the code written is always split first and foremost into host and device. Code written for the host is compiled on the Pynq board using the gcc compiler. Device code, also called kernels, that run on the ρ-VEX cores are compiled separately using ρ-VEX tool-chain and a rvexgcc compiler. The host-device setup is visualized in Figure 4.1.

In the following subsections this work will be explained in more detail with some comments on how to use the platform to setup a host program on the processing system's (PS) side that interacts with the dynamic ρ-VEX platform on programmable logic's (PL) side.

4.1 Host implementation

In developing code for the platform the first step hast to be writing code for the host. During this work the host was an ARM processor, but can, in theory, be replaced with any general purpose CPU that has access to an field programmable gate array (FPGA). The host program is responsible for setting up the ρ-VEX device, uploading kernels and starting programs, downloading bitstreams, monitoring if the stream is busy or not, allocating memory for the ρ-VEX to write to etc.

When creating a host program that interacts with the ρ-VEX there are number of object that have to be setup and configured. The first and most important is to initialize a context as without one there is no platform. Next step is to determine the desired number of ρ-VEX streams and how many cores there should be per stream. These constraints will be referenced time and again when setting up the host and device environment. Number of streams and cores can be changed later on while the program is executing and only a part of the initial setup is needed to be run

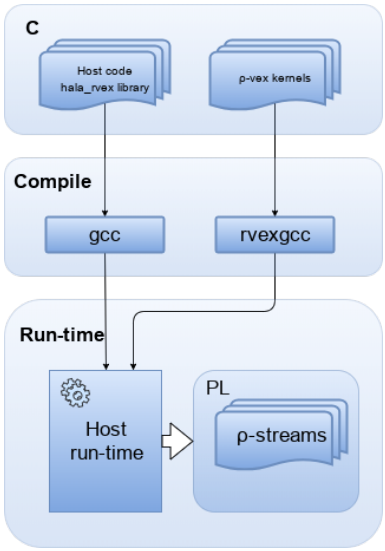


Figure 4.1: Overview of compile and runtime. Kernels and host code are compiled separately but host is responsible, at runtime, to configure the PL and upload kernels to streams.

again as will be further explained. The following steps include finding a compatible ρ-VEX device, downloading bitstreams to the reconfigurable fabric, uploading kernels to the ρ-VEX cores, configure and setup parameter, allocating contiguous memory space for the ρ-VEX to write back to etc. These concepts are discussed and explained in the following subsections.

4.1.1 Context

A `context` is an object that stores information needed for realizing the platform. When the `context` is initialized it opens a configuration file containing information about the platform and all available device setups. Contents of the configuration file are parsed to the context's variables and used for further setup of the host program. Values like the maximum number of available ρ-VEX cores on the platform, how they are memory mapped, how big the data- and instruction memories are and their offsets related to the ρ-VEX base address. The context also opens the `/dev/mem` file in the Linux host system and gives access via `mmap` to memory locations on the FPGA fabric. A pointer is created to an array that memory maps the whole ρ-VEX memory address space into a single array. Later in the process this array is used in slices to represent various memory locations of every core in each stream.

The information stored in the `context` is used when a platform is setup and when devices are configured. Setting up the platform and devices are the next steps in creating a host program and are further explained in the following sections.

4.1.2 Platform

Unlike `OpenCL` the `platform` object does not have a fixed device but only potential devices. They need to be setup via reconfiguration of the fabric, in line with the need and demand of an application developer. The idea is that every ρ-VEX stream should be an individual stand-alone device that can be setup and run independent of other streams/devices on the platform.

During the work of this thesis project this was not materialized. Only one device can be active at any given moment, but can be of any size, i.e., can contain multiple streams of equal size and running the same kernels. However, though only one device can be active on the platform at once, multiple devices can be kept in memory. If an application needs more, or less, processing power the reconfiguration time is reduced as bitstreams do not need to be read from file but can be sent directly from the host memory to the reconfigurable fabric. The `platform` object keeps a list of all devices that are in memory, along with a pointer to the binaries, and knows which one is active at any given moment.

4.1.3 Device

A `device` object holds all information about the device that has been created. It has access to the cores and streams on the fabric and is used for all communication to the actual ρ-VEX device. This object keeps track of the streams' states and knows for example if they are busy, running, down, programmed etc.

The ρ-VEX object is setup with the `initialise_rvex` function that takes an initialized context as a parameter along with a new ρ-VEX object and the desired number of streams and cores per stream. The context is used to see if the desired number of streams and cores are feasible within the platform. The device has to be initialized before any kernels or programs are uploaded to the device. During initialization, all streams to be used are added to the device along with all cores that each stream has in its pipeline. Every core gets a pointer to a memory-mapped array from the context to represent its data, instruction and register memories. The rule on how the ρ-VEX cores are mapped can be found in the documentation in Appendix A.

4.1.4 Reconfiguration

If an application requests a new device then the FPGA needs to be reconfigured. A bitstream is written to the `/dev/xdcfg` file that initiates a transfer to the configurable fabric. As partial reconfiguration (PR) was not implemented all data transfers on the PL side have to be stopped before a reconfiguration takes place. This means waiting for all ρ-VEX streams that are writing data to memory to finish as well as stopping all other direct memory access blocks (DMAs) or video DMAs (VDMAs) that might me running on the PL. If data streams are active at reconfiguration time it can lead to an unstable state of the reconfigurable fabric. A reconfiguration is needed every time an application requests more cores or a change in the streaming architecture.

4.1.5 Program

There is no function implemented that can take a program source as a parameter but only a function that creates or uploads precompiled binaries to the ρ-VEX cores instruction memory. When setting up programs on the host side the kernels to be used have to be precompiled and in binary format as is further explained in Section 4.2. The user calls the function `create_program_with_binary` that takes a program name and a device as parameters and downloads the right kernels to the correct cores' instruction memories. If the binary is bigger than the available memory, the function returns a **1** and prints out an error message containing the maximum binary size and the actual size of the program that was attempted to upload, otherwise a **0** is returned. The function `create_program_with_binary` does not start the kernels,

only uploads the instructions. Next step in the host program's implementation is to setup the ρ-VEX parameter.

4.1.6 Parameters

Communications between host and device go through the `parameters` object. This is a `struct` that holds predefined information as well as optional data parameters.

Parameters are transfered to the first ρ-VEX core of every stream. The way this was implemented is via a `transfer struct` that has few predefined variables that should not be changed by the user but are set up automatically when the `set_rvex_parameters` function is called. As the ρ-VEX has little endian memory system, every parameter that is written to the ρ-VEX has to be byte swapped. This is taken care of by the functions that write to the ρ-VEX and should not be of consideration to the end-user but is something that is worth knowing for future improvements of the platform.

As this is an image processing oriented design, the `struct` contains information about the image that is being transfered. The end-user is responsible for setting parameters such as total size of the image, its width, height and stride, along with information about the image block being uploaded in each iteration. This information contains the size of the rectangle part being uploaded, its offset in reference to the complete picture, its total width, height and pixel size. The last part of the transfer `struct` is the data to be uploaded. For other uses of the platform beside image processing, the data part of the `struct` can be used to upload any parameters or arrays as the programmer sees fit. Other fields specific for image information can be ignored or used to transfer other parameters.

4.1.7 Execution

When everything is setup, the device has been created, the kernels uploaded and the parameters configured it is time to start the ρ-VEX processors. First kernels are started by calling the `start_rvex_programs` function and using a ready `device` as a parameter. This function writes a control sequence to all cores' registers that start the program from the top.

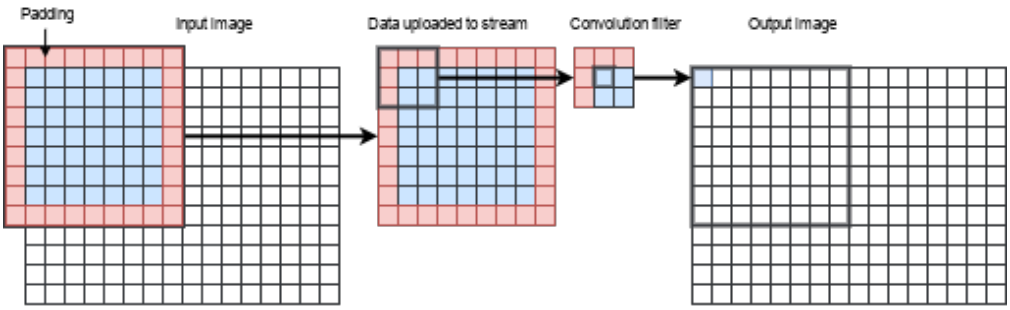


Figure 4.2: One rectangle is moved at a time, from input image to a stream. For convolution filtering, extra padding is needed around the rectangle.

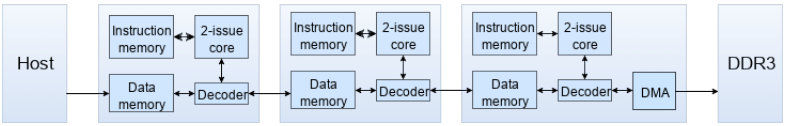
With the cores running, data needs to be moved to the device. To do this there are three functions already implemented. All have in common that `pthreads` are used to spawn threads, one per stream. A thread is responsible to send all data to a stream and split it up into smaller

chunks in case it is too big for the dedicated data memory of the processors. The simplest function writes lines to the first core of every stream. There are two functions, aimed at image processing, that write rectangles to the device (Figure 4.2). One splits the image into rectangles that fit into the data memory of the core that is being written to, while the other adds a padding around the image. This is done because convolution filters need a border around every pixel that is being processed as is shown in Figure 4.2.

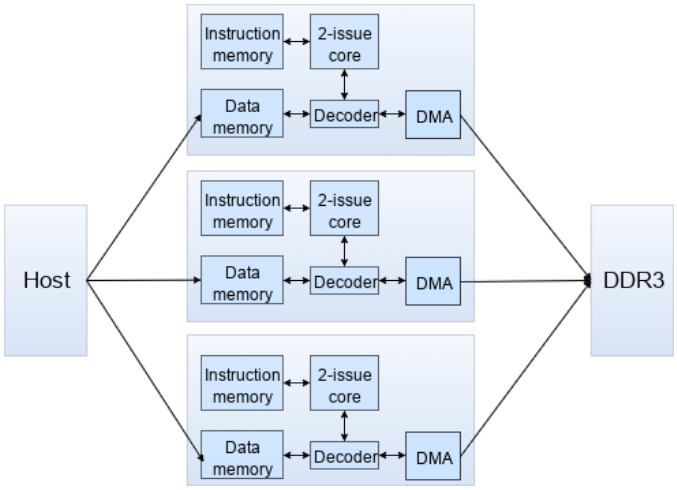
If the stream is busy when data is to be copied to the device the host program waits so pre uploaded data is not lost nor overwritten.

4.2 Device implementation

One of the core concepts of this platform is its dynamic hardware structure. Upon building an application for the platform the developer has great flexibility when it comes to device setup, total processing power, energy consumption etc. As discussed in Section 2.1 the device can be setup as a (1) manycore like a GPU shown in Figure 4.3b or (2) in a streaming fashion where one core outputs its results to the next one in line as shown in Figure 4.3a. Both setups have their advantages and disadvantages that will is examined in more detail later on.



(a) ρ-VEX cores lined up in a memory streaming pipeline.



(b) Overview of the platform setup where every core is independent.

Figure 4.3: Overview of host ρ-VEX cores can be lined up.

To understand the device setups better, we can take image processing as an example as it is often embarrassingly parallel, where each output pixel is processed independent of all other pixels in the output image. As has already been explained in Section 2.3, GPUs take advantages of this inherent parallelism by throwing in thousands of small cores, all doing the same job at

the same time but in different parts of the image. This can be done in a similar way with this platform although that is not its strong side. The interesting part lies in the streaming architecture where instruction parallelism is exploited along with data parallelism. As GPUs and the method of data parallelism is so widely adopted and well documented, this chapter will be focusing more on the streaming functionality of the platform.

The kernels running on the platform are written in C99 with few limitations related to the ρ-VEX compatibility. For example, the ρ-VEX version used in this project, does not have support for floating point operations nor a square root function which has to be taken into consideration when writing the kernels. The following section gives a high-level explanation on how programs are written for the ρ-VEX streaming platform.

4.2.1 Map algorithms to the streaming architecture

To understand how algorithms are mapped to the streaming platform we will keep using image processing as an example. Image processing pipelines, like the one shown in Figure 4.4, are inherently split into multiple stages, where the output of one stage is the input of the following one. They are a natural fit for the streaming architecture. At first glance, it might seem like a good idea to implement it on a stream consisting of three cores where each stage of the image pipeline is mapped to a single core. But this is not the case, as every stage in the figure has a different amount of arithmetic and memory operations. No automated tools or methods to map the image pipeline to the cores were made so it has to be manually.

Figure 4.4: A simple 3-stage image processing pipeline. First the input image is greyscaled, then blurred before a Sobel edge detection is applied and the final image is written to output.

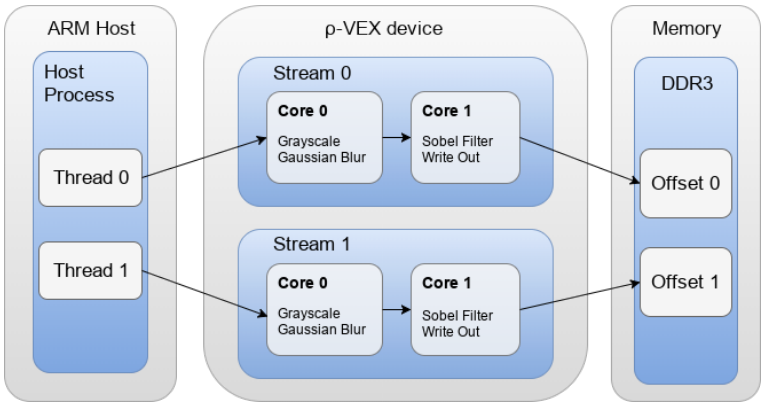


Figure 4.5: On the host side, one thread is spawned per stream. Here the gray-blur-sobel image processing pipeline has been mapped to two cores and two streams as an example.

Manual inspection of the edge detection pipeline reveals the number of operations in each stage, which are presented in Table 4.1. When the results are examined it becomes clear that a three stage pipeline would not be optimal as the last core, working on the sobel filter, would need

Table 4.1: Each stage of the image processing pipeline analyzed and divided into processor operations per pixel

Stage	ALU	Mul	Dev	Mem	Total
Grayscale	4	5	1	4	14
Gaussian Blur	21	6	1	10	38
Sobel Filter	33	4	0	13	50

to perform almost four times the operations on each pixel compared to the first core responsible for grayscaling the input image. A good fit would be to implement a two stage stream where the first core handles grayscaling and the Gaussian blur and the second core would be responsible for the sobel filtering for better resource utilization, leaving the system well load-balanced.

FPGAs have relatively small on-chip memory that sets constraints on the whole design. It is a limiting factor on the total amount of instruction memory available to each processor. If every core is to execute all stages of an image processing pipeline, i.e., every stream would consist of a single processor like in Figure 4.3b, there is not much space left for compile time optimizations, given that the binaries would even fit to memory. Techniques like loop unrolling can have significant impact on performance [2] at the cost of total binary size. This becomes possible when processors in the ρ-VEX stream run smaller part of the algorithm and is a great advantage of the stream architecture over the single core setup.

4.2.2 How cores communicate

As discussed earlier in Section 4.1, a developer who writes code for the platform can choose either having many independent cores or line the cores up into streams. Cores that are independent, can also be thought of as streams with a pipeline depth of one, have no means to exchange data. This is a limiting factor when using the platform in a similar setup as how GPUs are designed. GPUs normally have more complex memory hierarchies which are, for example, split into large global memory, smaller per block of threads memory and then a small private memory for each thread. When the ρ-VEX platform is run as a manycore it is limited to private memory only, with the exception of write access to host main memory.

When the platform is setup in a streaming fashion every core can write to its own memory, and with a flip of a switch, write to the memory of its adjacent core. The switch that is flipped is not more than a map where input memory starts at address 0x00000000 and the same address of the next core is mapped at 0x80000000. This means that when a function or a loop writes to a variable or array, it is up to the developer to choose if the data is to be written to the processor's own memory, or to the succeeding processor's memory. The following core can then start processing the data the moment it becomes available.

When kernels are written, it's very important to define a partially volatile transfer `struct` both at the input address of the kernel in question and for the adjacent kernel which outputs' will be written to. A flag that indicates when data is ready is externally updated and if that is not volatile, the compiler ignores all conditional statements regarding that flag as it is not updated with in the normal program flow. In the common header file that comes with the platform there are predefined transfer `structs` that can be used. They are used in the example programs, to correctly map memory of a core to the private memory of the next one in line. These `structs` have variables such as stream number, offset, state of the kernel, output address in host main

memory, data to be transfered and its size that need to be shared by two cores.

In total, there are three states a core can be in IDLE, READY and BUSY, indicating that core is not working, data is ready to be processed and data is being processed respectively. When the host starts sending data to the first core it marks the core as BUSY. After the data transfer has finished the state is updated to READY indicating that new data is ready to be processed. The core marks itself as BUSY so the data is not overwritten by the host while it is being processed. When results have been produced and written to the memory of the following core in the pipeline, that core can start processing the data immediately. The former core marks its state as IDLE when it is ready to receive new data batch and starts working on that dataset so both cores work simultaneously, but in different stages of the image processing pipeline.

4.2.3 Writing back to host memory

When the last core in a ρ-VEX stream has finished processing a line it needs to write it back to main memory. Like other cores in a stream, the last core has an external memory map to address 0x80000000. But in the case of the last one, this address maps to a DMA built into the streaming architecture. Writing to the DMA and subsequently to host memory is done in a number of steps with few things that have to be kept in mind when starting a transfer. Figure 4.6 shows the steps needed to initialize and start a DMA transfer.

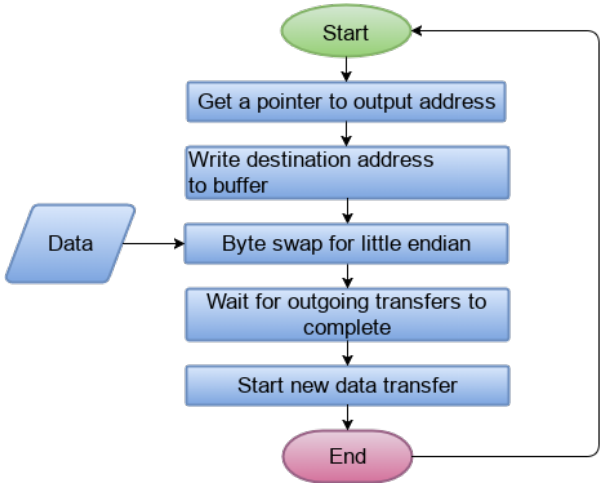


Figure 4.6: Functional flow overview on how DMA transfer is setup and initiated in the last core of every stream.

First, a buffer is created pointing to the output address 0x80000000. The destination address in host memory is written to the newly created buffer before it is filled with the data to be written back to memory. As ρ-VEX stores data in big endian format and the host, in this case an ARM, uses little endian convention, every integer has to be byte swapped before written to the output buffer. To start a new transfer, the size of the data array to be sent is written to the buffer. If there is any ongoing transfer this data will be ignored and subsequently lost. To avoid data loss, care has to be taken to wait for any previous transfers to complete before writing the data size to the buffer.

It is important to be very careful when writing back to host memory. The ρ-VEX DMA has no way knowing if the memory that is being written to is already occupied by a process

on the host system. It means that memory has to be allocated by some host process before the DMA is started to avoid instability. In worst case this could lead to a complete system failure if the DMA overwrites host-system memory. This can be avoided by allocating a dedicated part of the DDR memory to the ρ-VEX platform at Linux host-system boot time. This was not implemented in this work but what has been done to tackle this problem is a dedicated python server that uses Xilinx library to allocate consecutive memory blocks. There are few functions in the palloc library that communicate with the server and secure contiguous memory for the DMA to write to. To read more about the palloc library please refer to the platforms documentation in Appendix A.

4.3 Practical considerations

The platform implementation explained in this chapter was carried out on Xilinx Zynq processor running Ubuntu Linux with pre-installed Xilinx-Python libraries. To port the platform to another architecture all overlays have to be regenerated, leaving a big setup overhead for new platforms.

How communication from host to device was implemented, raises some security concerns. Super-user privilege is needed to access the host system `/dev/mem` file leaving the whole memory open for manipulation and can not be considered a good practice.

Though this implementation works it is far from production ready. It cans still serve as basis for further explorations into runtime reconfigurations as well as mapping algorithms to streaming architectures.

5 Experiments

In this chapter the focus is on evaluating the Hala ρ-VEX platform. It is important to measure and test the implementation developed during this thesis work. To evaluate and understand, how the platform fairs against other related work that was introduced in Section 2.5. A number of experiments were performed and in the following sections they are explained and the results from those experiments presented and discussed.

It is not easy to compare the Hala ρ-VEX streaming platform to other related work within the time frame of the thesis project, as it is out of scope to setup other related platforms and create benchmarks that fit all the various implementations. In this chapter the Hala ρ-VEX streaming memory platform is evaluated and tested. The focus will be on run-time reconfiguration, energy consumption, and image processing performance where streaming pipelines are compared to a stand alone core. Ease of programmability and design cycle time is also evaluated, but on a conceptual level, since it is hard to quantify. This chapter is concluded with a discussion section where the main results are evaluated and reflected on.

5.1 Setup and hardware

A Pynq board with a dual core ARM Cortex-A9 processor, that serves as a host, and a Xilinx 7-series FPGA for device implementation, as described in Section 2.2.1 is used for all experiments. All host code is compiled on the board and the kernels, that run on the ρ-VEX cores, are compiled on an x86 based laptop running OpenSuse Linux.

HDMI in and HDMI out is supported but that is mostly for demo purposes, so all experiments are performed by using an image loaded from memory. This both saves time, as setting up an HDMI data link does not happen instantly, and as the current HDMI setup is not ideal it shows better performance to use an image loaded from memory. Though the image is read from file, it is kept in memory when tests are performed to save fetching time and not read from the SD card in every iteration.

All results are based on real experiments performed on the Pynq board except for the power consumption results, that are generated using Xilinx's Vivado power estimation and the paragraph on theoretical performance that are based on Xilinx and ρ-VEX documentations.

5.2 Results

5.2.1 Reconfiguration

As has been explained in Section 2.2.2, the FPGA is reconfigured by uploading a new bitstream from the host processor to the reconfigurable logic. The steps needed to reconfigure the Hala ρ-VEX platform, besides uploading a new bitstream, is to download kernels to the uninitialized ρ-VEX processors. Few basic parameters need to be set and the programs need to be started.

According to Xilinx, the bitstream transfer rate is 400MB/s [29] for non-secure PL configuration and 100MB/s for secure PL configuration. This means that ideally it should take 10 or 40 ms to upload a new 4MB bitstream. For good measure this was tested on the Hala ρ-VEX platform by downloading a new bitstream, both from the SD card and directly from memory. To get a good average number it was uploaded 2000 times for both cases. It is not enough to just download a new bitstream, as stated earlier, so the time it takes to download kernels and start programs where also measured and also tested 2000 times. The results from these experiments are shown Table 5.1.

Table 5.1: Results from measuring average reconfiguration time of the Hala ρ-VEX platform.

Download bitstream from SD card to memory	16.5ms
Download bitstream to fabric from memory	180ms
Download kernels and start programs	0.15ms
Total reconfiguration time	196.65ms

The time it takes to download a new bitstream is a factor greater than what was expected. This could be improved by using Xilinx native drivers, and not rely on the Linux OS to move the binaries, but that is not confirmed. As can be seen in Table 5.1, the reconfiguration time can be improved just under 10% by keeping the bitstreams ready in memory and not fetch it from the SD card every time when reconfiguration is needed. This still leaves 180ms left to reconfigure the platform, so that any image stream working on higher frame rate than 5 frames per second (fps) would loose more than one frame during reconfiguration.

5.2.2 Performance

Performance can be measured in various ways, but as this thesis' focus is on image processing all of the performance tests are carried out by using image frames and image processing algorithms.

Hala ρ-VEX roof line model When measuring the performance of the Hala ρ-VEX platform it is useful to first know the theoretical maximum performance of the system. When it comes to performance of accelerators, bandwidth and number of operation per clock cycle are paramount. We have to know how much data can be moved to the accelerator per second and how many operations are possible in the same time interval, i.e., what is the maximum throughput of the system.

According to [5] the AXI4 protocol can write one word per clock cycle. The experimental setup used has an AXI4 bus clock set to 85MHz so maximum mount of data that can be moved should be

$$\text{Bus Bandwidth} = 32bits \times 85MHz = 2720Mb/s\,(340MB/s) \tag{5.1}$$

which can be translated to 236 fps in image processing, giving our test frame is 800 by 600 pixel color image.

The ρ-VEX cores work on the same frequency as the AXI4 bus. The maximum number of operations for a ten core setup would then be

$$\text{Peak Performance} = 85MHz \times 10cores \times 2lanes \times 1OP = 1700MOPS/sec\,(1.7GOPS/sec) \tag{5.2}$$

taking two lanes into account as in theory the ρ-VEX should be able to issue two commands simultaneously and complete one operation per clock cycle. Multiplication takes two clock cycles but can be pipelined so one result can be produced per clock cycle. Division takes longer, or around 40 clock cycles. The peak performance in Equation 5.2 is theoretical and can not be expected in a real application. The memory bandwidth, calculated in Equation 5.1, and peak performance are added together into a ten core roof line model shown in Figure 5.1.

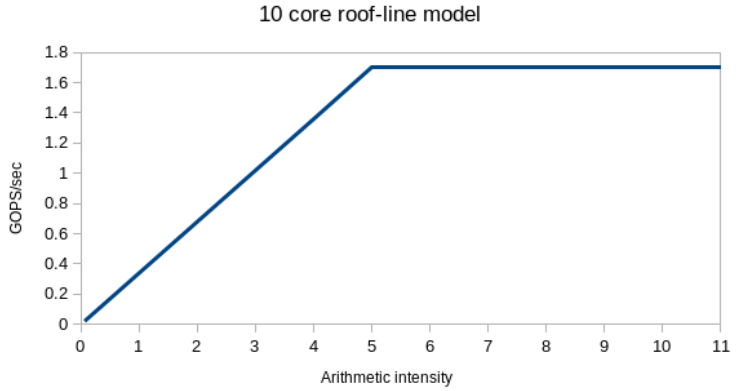


Figure 5.1: A theoretical roof line model of the ρ-VEX streaming platform, using 10 cores on a Xilinx Pynq development board.

Maximum performance is reached at five arithmetic operation per data byte. As image processing algorithms normally do more then five operations per pixel, it is safe to say that the memory bandwidth should not be a bottleneck in the `Hala ρ-VEX` platform, given the ten core setup. This roof line model is based on optimum data transfer performance as described in the Zynq book [5] and theoretical ρ-VEX peak performance.

Bandwidth When actual measurements are done on the `Hala ρ-VEX` platform the optimum bandwidth, calculated in the previous section, is not reached. The experiment was performed by repeatedly pushing a test image, an 800 by 600 colored image, to the accelerator without performing any operation on the pixels and not copy the image frame back to memory. The maximum measured bandwidth is only 18.5 fps which translates to 26.6MB/s. With a simple pass-through algorithm, i.e., the image is copied back to memory from the ρ-VEX cores, this performance is even lower, at 17 fps (24.8MB/s), as shown in Figure 5.2. The big difference between ideal performance and the one that is measured can be explained by the fact that in the `Hala ρ-VEX` platform the ARM processor is used to copy the data to the PL fabric. An ideal, and lot faster way to move the data would be to use a dedicated direct memory access block (DMA) to transfer images to the ρ-VEX cores. This was not done and therefore the ideal AXI4 bandwidth was not saturated.

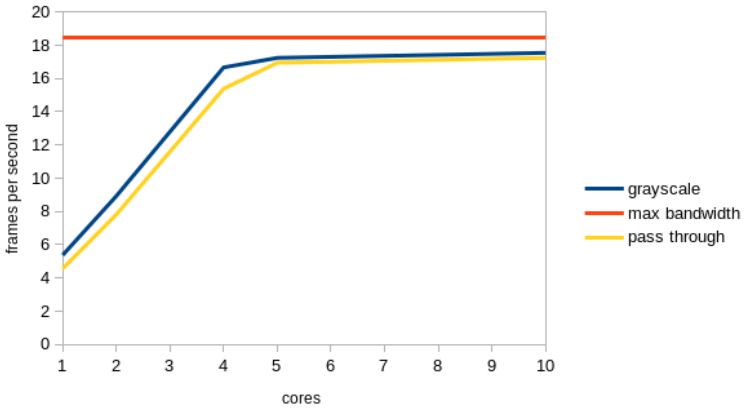


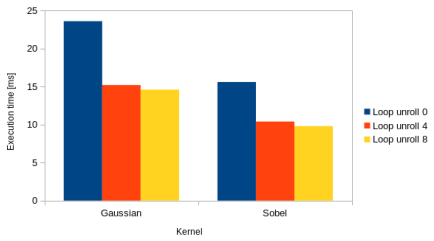
Figure 5.2: Throughput of 2 simple algorithms compared with the maximum measured bandwidth.

It is noticeable form Figure 5.2 that the grayscaling algorithm performs little better than the simple pass-through. The results are gathered by iterating the image transfer and grayscaling 1000 times to get a reliable average performance. The reason why grayscale was slightly faster is that when copying back to main memory, the image is three times smaller than the original one. In this example that means copying 480k bytes over 1.44M bytes, as with the colored image, per image frame.

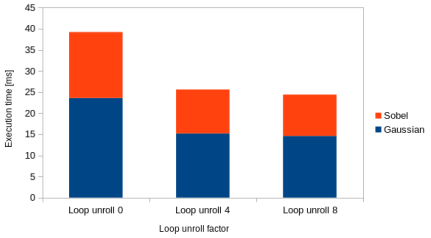
Loop unrolling One of the selling points of the streaming architecture is a better use of the instruction memory. By using loop unrolling, the kernel binaries grow in size, but at the same time the compiler can make a better use of filling the VEX instruction pipelines and ideally complete an execution of an operation every clock cycle.

Few experiments were done using loop unrolling factors of 0, 4 and 8. During the tests an 256 by 256 grayscale image was used and two different kernels were implemented. First a Gaussian blur kernel was tested and secondly a Sobel edge detection algorithm. The output from the blurred image from previous experiment was used as the input for the Sobel edge detection. To get a good average results the image was run 20 thousand times through the accelerator for both software kernels. The results are shown in Figure 5.3a and as can be seen, the performance was increased by around 60%, for both processing kernels, by unrolling the main loop 8 times compared to no loop unrolling at all. When execution times of both kernels are stacked together, Figure 5.3b, the total time goes down from 39.2ms to 24.4ms. The Sobel and Gaussian binaries roughly tripled in size by unrolling the main convolution kernel, or from 800B to 2kB and from 768B to 2.3kB respectively. As the instruction memory, per core, is only 4kB no further unrolling experiments were possible.

Streaming vs single core Another advantage of the memory streaming solution compared to independent many cores is image processing pipelines. Rather then waiting for all cores to run through all the algorithm before starting processing the next image in line, a part of the algorithm could be run in the first stage of the pipeline, and when finished, pass on the results on and start working on a new frame. By pipelining the processing procedure in this way the frame rate could be pushed up and overall throughput increased.



(a) Loop unrolling of Gaussian and Sobel kernels.



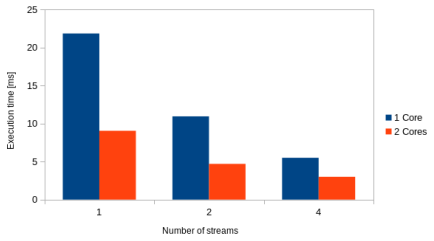
(b) Stacked processing times of Gaussian and Sobel kernels.

Figure 5.3: Results from loop unrolling two image processing kernels tested on two streams consisting of one core each.

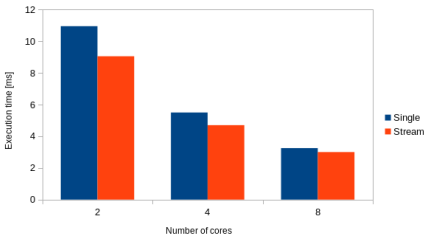
To evaluate the streaming architecture, with the limited resources of the Pynq board, two different streams are compared. A stream of just a single core is compared in performance with a stream of two cores. To get a better view of scalability, within the limited fabric, up to four streams are run in parallel. Two filters are used, a Gaussian blur and a Sobel edge detection.

In the first setup, both kernels are running on the same ρ-VEX processor. The intermediated results from Gaussian are stored in a local memory. When the intermediate result are ready, the ρ-VEX core signals the ARM to start a new data transfer while the edge detection algorithm runs. The benefit is that while the Sobel edge detection algorithm is being executed data is moved simultaneously to the cores data memory, i.e., the input data is refreshed.

The second setup splits the load between two separate processing cores. The first core applies the Gaussian blur and the second core is responsible for calculating the Sobel edge detection and writing back to host memory. As can be seen in Figure 5.3a there is some difference in execution time between the two kernels. The Sobel kernel executes in about two thirds of the time it takes to execute the Gaussian blur. In the streaming setup only the second processing core is responsible for writing data back to main memory. This should shorten the execution time of the Gaussian kernel and improve the overall load balancing of the stream.



(a) Results from running Gaussian and Sobel algorithms on streams of size one and size two.



(b) Total execution time compared between using streaming, and not streaming.

Figure 5.4: Single and dual cores streams compared using two separate image processing kernels. Less is better.

When the results are examined, there is considerable gain in performance using the streaming setup over the single core configuration. In Figure 5.4a the execution time between single core streams and two core streams are compared running the same image processing filters. It is interesting to see that the dual core streams perform over twice as good. For better clarity of resource utilization, in terms of performance per core, Figure 5.4b shows total execution time of both setups when equal number of cores are used. The dual core setup shows in all three

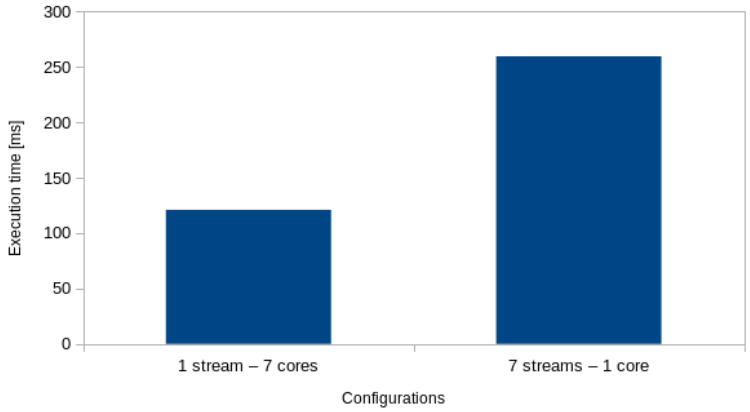


Figure 5.5: Average execution time of a seven stage image processing pipeline run on: (1) a single stream consisting of seven cores and (2) seven single core streams.

experiments better use of resources, i.e., shorter execution of the image processing pipeline per core.

Though the resources on the Pynq board are limited it is still possible to run a stream of seven cores on it. To test how a seven core stream fairs when compared with seven single cores, a seven stage image processing pipeline has been created. The algorithm developed was made to fit both for a stream and single core. It consists of seven processing kernels, beginning with a Gaussian blur, then there are five blur kernels of equal intensity, followed by a Sobel edge detection at the end. Both setups were tested using an image of size 800x700 and iterated 2000 times to obtain a good average execution time. The results from this experiment is shown in Figure 5.5.

The results from the seven core experiment show that running the same algorithm on seven single cores, more than doubles the execution time compared to running it on a single stream that consists of seven cores. The image processing pipeline used was made in such a way that the ρ-VEX processors in the stream are fairly equally load-balanced.

The reason for the difference in execution time can be explained by three separate factors. All the main loops in the streaming setup are unrolled 8 times and we have seen from earlier experiments that loop unrolling introduces a considerable performance gain. The second factor is number of pixels that need to be copied to the device per image frame. In the streaming setup all intermediate results are written to the next processor's memory, meaning that for every iteration, a 32KB slice of the original image can be uploaded from main memory to the first core. With the single core setup, there needs to be memory available for intermediate results. Two arrays are used to toggle results between image processing kernels. This means that only 10KB of the original image can be uploaded in every iteration. As there are two convolution filter in the pipeline the part of the image that is begin uploaded needs to be padded two times due to data dependencies of the convolution kernel as was explained in Section 2.3. As the image is split into more pieces, the overall data that needs to be copied increases as every iteration requires four lines of padding due to data dependencies. The third factor is related to the second one in the way that as the image is split into smaller pieces and padding is added for every upload, the total number of arithmetic operations per image frame increases as well. The streaming setup gains considerable performance with better compile optimizations, less data that needs to be copied to

the ρ-VEX device and fewer arithmetic operations per image frame.

5.2.3 Power consumption

The power consumption of the FPGA fabric is not measured in the field, but all numbers are based on Vivado power estimations. During power analysis some parameters are modified in order to help the tool to get better estimates.

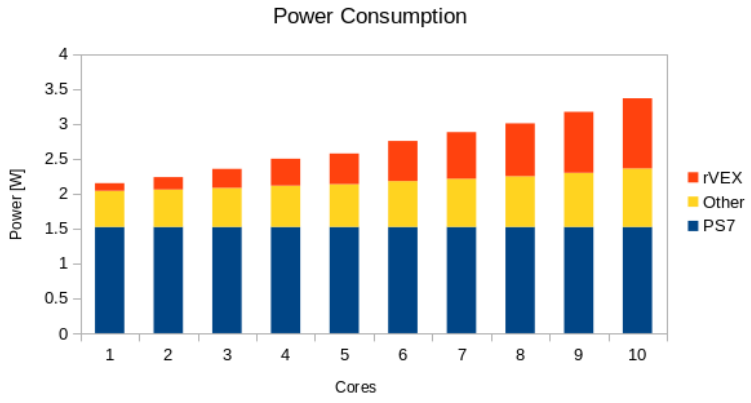


Figure 5.6: Total power consumption of the Zynq, SoC, showing near linear increase in power consumption with every new ρ-VEX core added.

In this experiment, the results are presented in Figure 5.6, it is assumed that the 2 ARM cores are working at 50% capacity. That there is no heat sink on the chip, no fan and that the ambient temperature is around 30C. What is interesting, is that even when the logic fabric is filled with 10 ρ-VEX cores, clocking at 90MHz, they still use less than half of the chip's total power consumption, or 1W. This was not tested on real hardware and are only estimations. If these estimation are not faraway from reality then the `Hala ρ-VEX` platform's performance per watt is around 1.7GOPS/W. Further real live measurements are needed to confirm if these results are anywhere close to real power consumption.

5.2.4 Programmability

It is not easy to measure how programmable, or how easy it is to program a device. One measure would be what language the application is written in, how popular is that language, and is it normally used for the stated purpose. For example, is it usual to write image processing algorithms in `C`?

The code, as explained in Section 4.1.3, is written in standard `C`. Though there are some things that have to be taken into consideration, like how the host communicates with the device, and how data and parameters are passed from one soft processor to another, processing kernels are still written in a fairly standard way for the `Hala ρ-VEX` platform. Image processing algorithms written for a more general CPU can be ported to the ρ-VEX platform without much modification. What has to be kept in mind though is the limit of the data and instruction memory sizes. There are some functions in the `Hala ρ-VEX` libraries that help break the dataset down into manageable portions that can be written to the device in order to not overfill the memory.

But fitting a big image processing kernel into the small instruction memory of the ρ-VEX is more of a challenge. So the code has to be split over several kernels manually.

There are examples in the `Hala ρ-VEX` platform directory on how an image processing pipeline can be broken up and shared between processors, and therefore make use of the streaming element of the platform. There is a learning curve for a developer in order to get started writing new code for the platform. But it should not be to alien to a programmer that is used to writing code for `OpenCL` platforms for example.

It is hard to say but it could take an experienced programmer, used to writing `C` code, about a day to get familiar enough with platform to be able to modify and compile one of the examples programs that come with the platform. To fully understand the platform and writing code from scratch would take longer. But when used to the architecture, a programmer can write new and experimental algorithms in a similar amount of time as with other platforms like `OpenCL`. This will soon be tested when the platform is publicly available.

5.3 Comparison

Though it is important to compare the platform to other similar works it is not a straight forward task. If we look at the high level comparison between other solution in Table 2.1, Section 2.5.5, then it would be fair, based on the experiments presented, to give `Hala ρ-VEX` a plus in programmability, a plus/minus in performance, two pluses in development cycle time and a plus in reconfigurability.

Based on the project goals `Hala ρ-VEX` comes out on top in total number of pluses. This might be an unfair comparison at the cost of the other platforms as not all their merits are counted up. If other things were taken in consideration then, for example, Catapult [6] scales over multiple FPGAs in data center, MARC [7] offers great performance close to that of a handcrafted hardware description code, LP-P^{2}IP [8] is high performance and runtime partial reconfigurable. Halide-HLS [26] offers a mixture of ease of programming with high performance at cost of long compiling cycles. All the solutions are good in their own way and `Hala ρ-VEX` is a fine addition to this group though it is still in its early development stage and not as mature solution as the others that were presented.

It would be interesting to test longer image processing pipelines, using more cores per stream, on bigger FPGAs for real performance.

5.4 Discussion

The experiments performed have shown that the platform is usable in most aspects. It is programmable, reconfigurable, can be used as a single core, many core, in a streaming fashion and running multiple streams simultaneously.

The outcome of the performance experiments are interesting and show good results in using the platform in streaming fashion over the multi core setup. The tests performed used rather simple algorithms and streams of only two cores. Still there is around 10% speed up in using the streaming architecture on the small FPGA on the Pynq board. When seven cores where tested, in both a streaming and many-core fashion, the streaming setup was twice as fast compared to the many-core setup. With a bigger FPGA it would quickly show diminishing return to increase

the number of cores, if every stream has only one core each. Then it would be interesting to see how performance, or overall throughput, can be increased by making the processing streams longer, i.e., increase the amount of ρ-VEX cores per stream.

The results from the power consumption experiments show that the device does not draw a lot of power. Power consumption per watt, as it was estimated, is greater then an AMD embedded graphics card [30] but also three times more than a high density multi GPGPU architectures [31] in a server environment. These where only estimation and are not confirmed by a real time experiment on an actual device.

Reconfiguration time is acceptable as long as the platform is used where temporary data loss is not an issue. If an application is critical to data loss the 180 ms reconfiguration time is not acceptable as it would loose 12 frames if a video stream is running at 60 fps.

These problems also have solutions as the platform is in its early stages. Reconfiguration time can be improved with partial reconfiguration where the data flow does not need to be stopped when the fabric is reconfigured. The scheduling of algorithms and load balancing can be improved to make better use of the streaming functionality. It would be interesting to test the platform in other application domains than image processing. A domain where the data flow is lower or where arithmetic intensity is higher.

Now that there exists a programming interface for the `Hala ρ-VEX` it is time for further experiments and studies. The platform can be ported to bigger FPGAs, tried in other domains and scheduling between cores can be experimented with.

6 Conclusion and future work

In this chapter the work presented in this thesis report is concluded. In Section 6.1 we will reflect on the results gathered in the previous chapter, look back at the original problem statement from Section 1.2 and see what has been achieved.

During this work many ideas for improvement and increased functionality arose. Some of these proposed improvements are presented in Section 6.2. We will have a look at proposed memory improvements, automation of mapping algorithms to streams, ρ-VEX read access to main memory and more.

6.1 Conclusion

In Section 1.2 this project's goal was introduced, along with what was to be achieved during the work presented in this thesis report. The goal was broken down to four vital steps that needed to be completed in order achieve the thesis goals. Here we reflect on the main goal and the steps needed to be completed, to see what was achieved, and already presented, in this thesis. The goal, as it was stated in Section 1.2:

Is it feasable to use the ρ-VEX memory streaming architecture as a general, programmable, run time reconfigurable image processing platform?

The results, presented in Section 5.2, show that the ρ-VEX memory streaming architecture [2] is runtime reconfigurable, it can be used to apply image processing algorithm on real time video streams and the platform is fully programmable. Streams of two cores showed better performance than running the same algorithm on two independent processing cores, indicating performance promise in future implementations using longer and more complicated algorithms on bigger FPGAs.

The problem statement was also broken down into four steps that needed to be completed in order to achieve this thesis goal. These steps, and what has been done are listed below:

1. *Create a general library in `C` that includes all necessary building block to interface a new application with the ρ-VEX streaming device.*

   To tackle this requirement the `hala_rvex` library was built, as was thoroughly explained in Section 4. It contains all necessary functions to develop a new application to run on the `Hala ρ-VEX` platform. The `hala_rvex` library combined with the functions implemented in `palloc`, the Python memory server and the `common` header file create a full interface to work within the `Hala ρ-VEX` environment.

2. *Implement a functionality so that the number of streams, and cores within a stream, can be dynamically changed at runtime.*

This problem was solved by pre-synthesizing all possible ρ-VEX processing configuration that can fit on the programmable logic (PL) present on the Pynq board. As can be seen in Table 3.1 there is a total number of 27 possible lineups of the cores within the Zynq's PL. With every layout synthesized, and stored on the Pynq board, they take up a total space of 108MB. The Pynq's main storage is a micro SD card. A modern SD card, at the time this thesis is written, is available with hundreds of GB in data storage capacity. All 27 bitstreams only use a fraction of the available theoretical storage on the board.

This solution scales even though bigger FPGAs require bigger bitstreams and the number of possible setups grow exponentially. The whole bitstream library can be stored on any external storage space. If the number of setups needed by an application is known at compile time then all bitstreams can be fetched as a part of the application setup process. The bitstreams are then stored in memory for faster reconfiguration time. The bigger the FPGA used, the bigger will the bitstream be, the time it takes do load a bitstream should be proportional to its size.

3. *Shorten the application development cycle by removing the need to create a new hardware design while new program is being developed.*

   The `Hala ρ-VEX` platform is fully programmable at the post synthesis state. The kernels, for the device, are written in plain `C` as well as the host program. All possible hardware configuration are already synthesized and ready at runtime. It should only take as long to develop a new application for the platform as for any other hardware accelerator, i.e., in depth hardware design knowledge is not needed. In fact there is no need to run any type of hardware design tool in order to develop and run a new application for the platform.

4. *Create an input output interface for video streaming to properly showcase the platform.*

   A hardware design, that supports both HDMI input and HDMI output, was introduced in Section 3.3. Though it is not without flaws, there exists a demo that takes input from HDMI, routes the data to the ρ-VEX streaming device and back to HDMI out. In the current implementation there is great overhead of data movement, pixels being copied back and forth.

   The goal of interfacing the platform with input and output video data was met but is in a need of a new design. In future work, Section 6.2, some suggestions are made in how it can be improved.

What we have is a working platform. Device and host code has been mostly decoupled, but further work is needed and will be explained in the following section. The result of this work is a functioning open source platform that has been uploaded to the ρ-VEX online repositories. It will be available to anyone that wants to make use of it. In order to start using the platform, only thing that a user needs is a Pynq board and a computer running Linux operating system in order to compile new kernels for the ρ-VEX cores. The platform is still work in progress and needs to be further developed on both the hardware and the software level. Some suggestions are made in the following section, future work.

6.2 Future work

There is room for improvement in the platform's implementation and many things that would have been great to implement better or differently during this thesis work. Possible improvements can be found on all levels of the platform. In the VHDL code, in the Vivado design and in some C level functions. Few flaws especially stand out and they will be covered in the following paragraphs, in no particular order, with some recommendation on how they can be fixed.

Partial reconfiguration A prominent feature for the platform is to make use of partial reconfiguration (PR) [32]. By implementing PR it is possible to add new streams to the fabric without stopping other streams, and other functionality, running on the board. PR is not only faster than full reconfiguration, but a video stream can flow uninterrupted through the fabric while the change to the hardware is being made.

PR can take the Hala ρ-VEX platform to a new level with regards to dynamic run time reconfiguration and uninterrupted uptime. Another benefit that comes from using PR is storing all possible configurations. PR bitstreams are considerably smaller then full bitstreams. With bigger FPGAs both the number of available configurations increase and each bitstream increases in size. This means that more storage space is needed and it takes longer to download each bitstream to the programmable fabric. PR reduces these two overheads with smaller partial bitstreams replacing full bitstreams.

Memory bandwidth In the current implementation of the Hala ρ-VEX platform memory bandwidth is one of the biggest bottlenecks. There are numerous ways to increase the overall throughput and most of them include decreasing data copying and make better use of solutions already available.

With regards to streaming video data, it would be ideal to stream the pixels straight to the ρ-VEX streaming device. The overhead of first writing to memory and then to the ρ-VEX streams is far to great. It includes writing to a off-chip memory only to be copied back. How the video data can be streamed directly to the ρ-VEX needs to be further researched as there are few things that have to be kept in mind. For instance, if the incoming frame rate is higher then the ρ-VEX streams can process, should there then be a frame buffer so that only whole frames are dropped and not just parts of frames, and what will split the lines between ρ-VEX streams. With this in mind there could be an IP block implemented that takes care of this logic: stores one frame at the time and equally splits it between number of streams available.

If main memory is to be used for storing frames, there needs to be a better way of moving data to the ρ-VEX streams. In the current implementation the ARM processor copies all data to the ρ-VEX device. There are other ways to move the data from the main memory to the accelerator, like a dedicated direct memory access (DMA) block for example, and they should be further explored.

ρ-VEX access to main memory In the current design, when the ρ-VEX streaming device needs to write back to memory a dedicated Python memory server has to be started up first. This server is responsible for allocation of continues memory block for the ρ-VEX DMA to write to. This is not very practical and a better solution is needed. One way to tackle this shortcoming

of the platform, is to reserve part of the DDR memory for the ρ-VEX at boot time. I.e., hide considerable amount of memory from the host operating system.

In Section 4.1 it is explained how pixels, and images, are copied to the ρ-VEX device. The accelerator operates as a slave that can only wait for data and not request it. The last core in any stream can write to main memory and in similar way the first processing core of every stream should be able to read from the main memory. In order to materialize this functionality the VHDL code representing the streaming design would need to be revisited.

ρ-VEX access to main memory requires further research into possible overhead of such a solution in terms of resources and how it affects load balancing of the device. In the current implementation, in theory, there is no downtime of the first processing core. It can work on parts of the input data simultaneously as data is being moved to the core, initiated by the host processor. If the ρ-VEX would be responsible for accessing data in main memory, would that then stall the processor while data is being moved, for example.

Mapping an algorithm to a stream In the current implementation, represented in this report, algorithms are mapped to streams manually, as explained in Section 4.2. Mapping big complicated image processing algorithms has not been tested and it is not obvious how easily it will scale. In the current work, stages of the image processing pipeline are tested individually to see how long they take to execute in order to improve load balancing between various stages in the ρ-VEX processing pipeline. With long and complicated algorithms this could become a very tedious job.

It would be interesting to study how algorithms can be automatically mapped to the streams. To automate a process that detects where an applications can be ideally cut and split between processors. Each processor should get a similar workload to avoid cores running without performing any work.

Automating the procedure of mapping algorithms to the streaming architecture should be done in steps. First, there needs to be some improvement in how cores, within a stream, communicate data. As was presented in Section 4.2, data is moved from one core to another via memory map and dynamic structs directly exposed to the application developer. This would need to be improved and the underlying functionality further abstracted. A processing core could, for example, be called via asynchronous function call. This implies that code for all cores can be written within a single function, or a kernel. With the code for the whole stream being written within a single function it would be easier to create an automated process that analyses the data flow and identifies possible places where to cut the code and split it between ρ-VEX processors. The third step is to automatically create the individual kernels from the code snippets cut by the analyzing algorithm. This requires further studies and experiments before a good solution is found and implemented.

Variable instruction and data memory sizes In Section 3.3, we saw that block RAM (BRAM), the building blocks for instruction and data memory, is a limited resource on an FPGA. In the current implementation, data memory for every processor is set to 32kB and instruction memory to 4kB. This setup fully exhausts all BRAM resources on the Pynq board when ten ρ-VEX cores are implemented.

For image processing, it is good to have large data memories at the cost of smaller instruction memories. This might not be the ideal for other use cases. If the platform is used for different

applications, with fewer data points and more complicated algorithms, it might be of interest for an application developer to have bigger instruction memories and smaller data memories.

To create a bitstream for every possible memory setup within every possible ρ-VEX core setup does not scale very well. So a different type of solution is needed. This could maybe be solved by using PR, but that might also not be the case. Finding a way to realize variable memory sizes in the post synthesis state is an interesting work to look into.

OpenCL implementation To make the platform OpenCL compatible would be a great addition to the family of OpenCL ready devices. OpenCL, as stated in Section 2.4, is an industry standard in image processing and other GPGPU applications. There exists a great variety of algorithms, applications and documentation for OpenCL.

If the `Hala ρ-VEX` architecture would be OpenCL compatible, it would be fairly easy to compare it to other devices on the market. It would also minimize the learning curve for experienced OpenCL programmers, and the ρ-VEX streams could work alongside other accelerators within the same application.

The amount of work needed to be performed to make the device OpenCL compatible is not known. The ρ-VEX processor has been used as an OpenCL accelerator as part of the portable computing language (POCL) [33] and the ALMARVI project [24]. The body of the work would be to map OpenCL kernels to streams and the scale of such an undertaking is unknown at the time when this is written.

Bibliography

[1] J. Hoozemans, J. van Straten, and S. Wong, "Using a polymorphic vliw processor to improve schedulability and performance for mixed-criticality systems," in *Proc. 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hsinchu, Taiwan, August 2017.

[2] J. Hoozemans, R. Heij, J. van Straten, and Z. Al-Ars, "Vliw-based fpga computational fabric with streaming memory hierarchy for medical imaging applications," in *Proc. 13th International Symposium on Applied Reconfigurable Computing*, Delft, The Netherlands, April 2017, pp. 36–43.

[3] "Pynq - python productivity for zynq - home," http://www.pynq.io/, (Accessed on 01/22/2018).

[4] L. Crockett, R. Elliot, M. Enderwitz, and R. Stewart, *The Zynq Book: Embedded Processing Withe ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC*. Strathclyde Academic Media, 2014. [Online]. Available: https://books.google.nl/books?id=9dfvoAEACAAJ

[5] "Zynq-7000 all programmable soc technical reference manual (ug585)," https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, (Accessed on 02/01/2018).

[6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015.

[7] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, "Marc: A many-core approach to reconfigurable computing," in *2010 International Conference on Reconfigurable Computing and FPGAs*, Dec 2010, pp. 7–12.

[8] Á. Avelino, V. O. Roda, N. Harb, C. Valderrama, G. Albuquerque, and P. D. C. Possa, "Lp-p2ip: A low-power version of p1ip architecture using partial reconfiguration," in *ARC*, 2017.

[9] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," vol. 14, 10 2016.

[10] E. Houtgast, V. M. Sima, and Z. Al-Ars, "High performance streaming smith-waterman implementation with implicit synchronization on intel fpga using opencl," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, Oct 2017, pp. 492–496.

[11] 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.

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

[13] S. Wong, T. van As, and G. Brown, "r-vex: A reconfigurable and extensible softcore vliw processor," in *2008 International Conference on Field-Programmable Technology*, Dec 2008, pp. 369–372.

[14] 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.

[15] "7 series fpgas data sheet: Overview (ds180)," https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, (Accessed on 02/05/2018).

[16] "Axi reference guide," https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, (Accessed on 02/05/2018).

[17] "Cortex-a9 technical reference manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf, (Accessed on 02/05/2018).

[18] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601174

[19] "Opencl overview - the khronos group inc," https://www.khronos.org/opencl/, (Accessed on 02/08/2018).

[20] "About cuda | nvidia developer," https://developer.nvidia.com/about-cuda, (Accessed on 02/08/2018).

[21] "Intel fpga sdk for opencl - overview," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, (Accessed on 02/08/2018).

[22] M. Hosseinabady and J. L. Nunez-Yanez, "Optimised opencl workgroup synthesis for hybrid arm-fpga devices," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–6.

[23] "https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf," https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf, (Accessed on 03/08/2018).

[24] "Almarvi 621439," http://www.almarvi.eu/#publication, (Accessed on 02/11/2018).

[25] N. Asadi, C. W Fletcher, G. Gibeling, J. Wawrzynek, W. H Wong, and G. Nolan, "Paralearn: a massively parallel, scalable system for learning interaction networks on fpgas," 07 0002.

[26] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462176

[27] "Xilinx/pynq: Python productivity for zynq," https://github.com/Xilinx/PYNQ, (Accessed on 02/27/2018).

[28] "https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_vc707_eval_bd.pdf," https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf, (Accessed on 02/28/2018).

[29] "Xilinx partial reconfiguration of a hardware accelerator on zynq-7000 all programmable soc devices (xapp1159)," https://www.xilinx.com/support/documentation/application_notes/xapp1159-partial-reconfig-hw-accelerator-zynq-7000.pdf, (Accessed on 02/09/2018).

[30] "New amd gpu delivers up to 3x performance-per-watt for low-power embedded applications," https://www.amd.com/en-us/press-releases/Pages/new-amd-gpu-2017oct03.aspx, (Accessed on 02/26/2018).

[31] Y. Gao, S. Iqbal, P. Zhang, and M. Qiu, "Performance and power analysis of high-density multi-gpgpu architectures: A preliminary case study," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, Aug 2015, pp. 66–71.

[32] "https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf, (Accessed on 03/01/2018).

[33] "pocl/pocl: pocl - portable computing language," https://github.com/pocl/pocl, (Accessed on 03/01/2018).

A

This appendix contains a copy of chapter 2 from the Hala ρ-VEX platform manual as it was at the time of this writing. The manual does not cover all functions available within the platform libraries but all functions that are used in the demo application are documented here.

2 | Hala ρ-VEX API

This chapter gives an overview of the `Hala` ρ-`VEX` platform's API. An applicatoin that runs on the platform is split into host and device code. The host is an ARM processor and a device is the ρ-VEX streaming architecture.

The first section of this chapter deals with host code. It explains, in steps, how to write new code for the host side of the application and gives a simple code example. The second section is about how to write code for the ρ-VEX device with some examples given. The third and the fourth sections list functions used to setup a new application on the platform. Section three is a list of `hala_rvex` functions and in section four is a list of `palloc` functions.

2.1 Host code

Host code is written in `C` and compiled using `gcc` on the ARM processor. The host is responsible for setting up the ρ-VEX device, HDMI support and memory allocation. The host splits images to smaller pieces and sends data to the device.

The first section explains how to write new host code and the second section gives an example of host setup code.

2.1.1 New application

This section explains the basic steps that need to be taken in order to write new host code. When writing a new application the `hala_rvex` header file needs to be imported as it contains all necessary function to interface with the platform. If the ρ-VEX device is to write results back to memory then the `palloc` header file needs to be imported as well.

A `context` object has to be created before other things are initialized. The `initialise_context` function opens up a configuration file and reads core parameters into the `context's` attributes. Second step in writing a host application is to read a bitstream from file, the function `read_bitstream_from_file`

is used to read a bitstream from file and into memory. It takes number of streams and number of cores as parameters along with a bitstream *char pointer. There is a separate function, `download_bitstream`, that downloads the bitstream to the fabric. The reason these are two separate functions is that it might be of interest to the developer to keep more than one bitstreams in memory for a faster, on demand, reconfiguration.

After the bitstream has been downloaded there are functioning ρ-VEX cores on the fabric ready to be initialized. The initialization is mostly about memory mapping the streams and cores memory offsets as they are described within the `context's` attributes. This is done with help of the `context` object and the information about the available device is stored in a `rvex` object. With an initialized `rvex` object it is now possible to upload kernels to the devices. The function, `create_program_with_binary`, takes the device, context and name of the kernels as parameters. The convention is that all kernels are precompiled binaries with the same name. The kernels should have a suffix indicating what their position within a stream. If there is only one core within a stream its suffix would be zero. Same goes for the first core of every stream. Second is one, third is two etc.

The ρ-VEX streaming device writes straight back to main memory via a DMA. Before the device is started a continues memory block needs to be allocated. A call, via the `palloc` interface, to the Python server asking for memory takes care of that.

The device has to be prepared by calling the `set_rvex_parameters` and passing information about the input and output images' as parameters. At this stage everything is setup and all that is left is to call `start_rvex_programs` and write data to the device via `write_padded_lines_to_rvex`.

Next section gives an example of a host program.

2.1.2 Host code example

Below is a simplified example of a complete program that runs on the host. What is different from the program described in the previous section is that this application calls the `palloc` to start VDMA transfers. This means that HDMI is activated and the device will write to a memory allocated by the VDMA block. The application ends with stopping VDMA. If this is not done before the fabric is reconfigured it will it will go into an unstable state and the board needs to be restarted.

```
int main(){

  initialise_context(&ctx);
```

```
    read_bitstream_from_file(&ctx,..,..,..);

    download_bitstream(bitstream, bitstream_size);

    start_VDMA();

    initialise_rvex(&ctx, &device,..,..);

    create_program_with_binary(&ctx, &device, "streaming-sobel");

    get_input_address_from_VDMA(&input_address);

    get_output_address_from_VDMA(&output_address);

    set_image_properties(&input_image,..,..,..);

    greyscale_image(&output_image, input_image);

    set_rvex_parameters(&device, output_address, input_image, output_image)

    start_rvex_programs(device);

    write_padded_lines_to_rvex(&device, input_image);

    stop_VDMA();

    return 0;
}
```

2.2 Device Code

Code written for the ρ-VEX processors is written in `C` and follows the same principles as writing code for any other ρ-VEX implementation. The instruction memory of the cores is very small, only 4KB, so that has to be kept in mind when writing new kernels.

The kernels communicate via shared memory, i.e., one core has access to another adjacent core via memory map. The start address of data memory for any core is **0x00000000**. The same address for the next core in line, seen from the previous one is **0x80000000**. A core has both read and write

access to the next core's memory space.

The `common` header file needs to be included in all device applications. It includes a struct that contains attributes such as a core's states, output address, data sizes and data arrays. It is important at the start of every core to define a new transfer struct for both input and output:

```
#include "common.h"
    transfer *in = (transfer *) INPUT_MEM;
    transfer *out = (transfer *) OUTPUT_MEM;
```

The `common.h` also includes three stages a processing kernel can be in: IDLE, BUSY, READY. If a kernel is IDLE another kernel, or the ARM host, can upload a new set of data to it. If it is in READY state, it means it can start processing data. If BUSY, no data should be written to it as it could cause loss of data that has not been processed yet.

Below is an example of device code that could work for a core anywhere in a streaming pipeline except for the last core. The last core has few additional programming steps as it needs to initialize a DMA and write data back to memory.

```
#include "common.h"
int main() {
  transfer *in = INPUT_MEM;
  transfer *out = (transfer *) OUTPUT_MEM;
  volatile char *out_state;
  #pragma unroll(0)
    while (1) {
          if (*state == READY) {
            in->state = BUSY;
            out->out_address = in->out_address;
            for (int pixel = width; pixel < last; pixel++) {
                  // Applay kernel
            }
            out->offset = in->offset;
            out->state = READY;
            in->state = IDLE;
          }
        }
}
```

Some things were left out of the code to make it shorter, like the main filter kernel inside the main for-loop. What is important here is that when new data has arrived to the processor it marks itself as BUSY. Then all pixels are processed, and the state of the next core is marked as READY, signaling new data has arrived. The core then marks itself as IDLE and is ready for new data from the ARM host.

The last kernel of every stream has few additional steps as it is responsible for writing back to main memory. The last core has a built-in DMA that is mapped to **0x80000000**. Instead of showing the whole code, that looks mostly like the one from the previous core, below is a code snippet on how to write back to memory:

```
out_size = 255;
volatile int *buf = *(int *volatile *) 0x80000000;
int addr = in->out_address;
*buf++ = addr;
for(pixel = 0; pixel < out_size; pixel ++){
  // Fill the buf array with data
}
buf = (int *) ((int) buf & ~0x7F);
// Wait for other transfers to finish
while (*((volatile int *) OUTPUT_MEM) & (1 << 12)) {};
// Start the transfer.
*buf = out_size;
```

The DMA starts transferring data to the output address after maksing out the start address and writing the transfer size to it. The transfer size is in number of integers and has a maximum size of 1024 bytes per transfer.

2.3 Functions in hala_rvex.h

Listed below are the functions used by the demos provided with the platform at the time of writing. There are more functions within the `hala_rvex` header file and will hopefully be documented here in the near future.

2.3.1 initialise_context

Type	Parameters
Int	context *ctx

A `context` is an object that stores information needed for realizing the platform. When the `context` is initialized a configuration file is opened that contains information about the platform and all available device setups. Contents of the configuration file are parsed to the context's variables and used for further setup of the host program. The `config.conf` is stored in a conf folder and contains:

- Available number of streams

- Available number of cores
- Core memory offset in relation to a stream
- Stream memory offset in relation to the ρ-VEX base memory address
- ρ-VEX base memory address
- Data memory size per core
- Instruction memory size per core
- Data memory offset in relation to a core
- Instruction memory offset in relation to a core
- Creg offset in relation to a core

2.3.2 read_bitstream_from_file

Type	Parameters
Int	context *ctx
	int number_of_streams
	int number_of_cores
	char **buffer
	size_t *size

This function reads a bitstream from a file and into memory. The bitstreams need to be be in a `../overlays/` folder and the naming convention is `rvex_<number of cores per stream>_<number of streams>`. The function takes an initialized context as a parameter, desired number of cores and streams, and a pointer to a buffer and size. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.3 download_bitstream

Type	Parameters
Int	char *buffer
	size_t size

This function takes a buffer pointing to a bitstream and downloads it to the FPGA. This function needs root access as it opens `/dev/xdevcfg`. Returns zero upon success. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.4 initialise_rvex

Type	Parameters
Int	context *context
	rvex *rvex
	int number_of_streams
	int number_of_cores

This function initializes a new rvex device. It needs an initialized `context` as a parameter, a long with number of streams, and cores within a stream, that are on the fabric. The `rvex` device is memory mapped according to the context parameters. The function needs root access as it uses `mmap` to open the `dev/mem` file to access the operating system's memory. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.5 create_program_with_binary

Type	Parameters
Int	context *ctx
	rvex *rvex
	char program[]

This function uploads precompiled kernels to the ρ-VEX cores. The `context` and `rvex` objects need to be initialized before this function is called. The `program` parameter is the name of the kernel that is to be uploaded. The naming convention for the kernels is any name plus a suffix indicating the number of a core within a stream. The suffix should not be added to the `program` parameter. Example:

A Gaussian blur pipeline that works on a stream of two cores is called "*gaussian*" then there needs to be two binaries named "*gaussian0.bin*" and "*gaussian1.bin*" that will be uploaded to kernel-0 and kernel-1 of the ρ-VEX stream respectively.

2.3.6 set_image_properties

The struct `image_info` is used as a wrapper for parameter related to an image that is to be processed. For example, image size, width, height, stride, if it is padded or not and how big slices are to be uploaded to the device per

Type	Parameters
Int	image_info *ptr
	unsigned char *data
	int width
	int height
	int pixel_size
	int streams
	int padding
	int rect_height
	int rect_width
	int number_of_rects

iteration. This function is a setter for the `image_info` object. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.7 grayscale_image

Type	Parameters
Int	image_info *grey_image
	image_info input_image

This is a simple function that takes an initialized `image_info` object and returns a copy but with all color dimensions set to 1 no matter what the original dimensions were. This is helpful when an image that is to be uploaded will be grayscaled by the image processing kernels. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.8 set_rvex_parameters

Type	Parameters
Int	rvex *device
	int address
	image_info input_image
	image_info output_image

This function preparers the ρ-VEX devices for the task that is to be executed. It uploads few important parameters like stream number, stream state, dimensions of the image that is to be processed and the memory address that the ρ-VEX devices write back to. The memory space that `address` refers

to has to be allocated before the device is started. A memory server written in Python, and accessible via `palloc.h` is suitable to allocate continues memory space for the device to write back to. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.9 start_rvex_programs

Type	Parameters
void	rvex device

This is simple function that starts all programs, already uploaded, on all ρ-VEX cores that are initialized in the `rvex` device object.

2.3.10 write_padded_lines_to_rvex

Type	Parameters
Int	rvex *device
	image_info image

This function does nothing but spawning as many threads as there are streams. Every thread calls the `write_padded_lines` function. It waits for all threads to finish executing before returning. Returns *0* if successful and prints out an error message if something goes wrong.

2.3.11 write_padded_lines

Type	Parameters
void *	void * arg

This function is called from `write_padded_lines_to_rvex` and writes data to the ρ-VEX streams. The parameter is expected to be a pointer to a `rvex_stream`. The stream should already be initialized with image information and data. This function writes data to the stream according to predefined number of lines and adds pixel padding if requested.

Before uploading a data chunk to a stream, a check is made to see if the stream is busy, if busy, the thread waits a micro second and tries again.

Type	Parameters
Int	const char *file
	int *width
	int *height
	int *pixel_size

2.3.12 get_image_dimensions

This is one of three function that help with opening images as data files. The image has to be already split into to separate files by the image converter found in the tools folder. This function takes the name of the image and searches for a file with that name and the file extension *.dim*. The dimension file should include image width, hight and pixel size. The function writes the dimensions to the provided parameters.

2.3.13 get_image_from_file

Type	Parameters
Int	const char *input_file
	unsigned char *image_data
	long size

This function reads image data from file which is name is provided by `input_file`. The image data array has to be allocated before passed to this function and size has to be known. Returns *0* if successful, else a *1* and prints out an error message.

2.3.14 write_image_to_file

Type	Parameters
Int	context ctx
	int address
	long size
	const char *file

This function is used to retrieve image data from a specific memory location. This is very helpful when the device is used for processing images as the ρ-VEX streams write straight to memory. The function takes a `context` as a parameter to access the system memory file `/dev/mem`. It reads the content

of the memory location to a file and adds *.out* to the file name provided as a parameter.

2.4 Functions in palloc.h

Palloc is an interface to a Python memory server that should be started before any application is started for the platform. The Python memory server is responsible for starting and stopping VDMA transfers on the programmable logic and to allocate continues memory spaces for the ρ-VEX devices to write back to.

2.4.1 get_output_address_from_VDMA

Type	Parameters
Int	int *address

This function calls the Python memory server and asks what address the VDMA is reading from when streaming data from memory to the HDMI output port. Returns *0* if successful and prints out an error message if something goes wrong.

2.4.2 start_VDMA

Type	Parameters
Int	void

Asks the Python memory server to start HDMI input and output channels. Returns *0* if successful and prints out an error message if something goes wrong.

2.4.3 stop_VDMA

Type	Parameters
Int	void

Asks the Python memory server to stop HDMI input and output channels. This function has to be called before the logic fabric is reconfigured or the system becomes unstable. Returns *0* if successful and prints out an error message if something goes wrong.

2.4.4 get_input_address_from_VDMA

Type	Parameters
Int	int *address

Makes a request to the Python memory server for a memory address that the VDMA is writing to from the HDMI input port. Returns *0* if successful and prints out an error message if something goes wrong.

2.4.5 free_buffers

Type	Parameters
Int	void

It is important to ask the server to free all memory that has been allocated. This only happens if this function is called or the Python memory server is stopped. Returns *0* if successful and prints out an error message if something goes wrong.

2.4.6 get_memory_address

Type	Parameters
Int	long size
	int *address

Sends a request to the Python memory server for a new continues memory space of size `size`. The new address is written to the variable pointed to by `*address`. When the memory is no longer needed it is important to call the `free_buffers` function to free the memory again. Returns *0* if successful and prints out an error message if something goes wrong.