Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2015

MSc THESIS

An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL

Hugo van der Wijst

Abstract

In recent years the use of co-processors to accelerate specific tasks is becoming more common. To simplify the use of these accelerators in software, the OpenCL framework has been developed. This framework provides programs a cross-platform interface for using accelerators.
The ρ-VEX processor is a run-time reconfigurable VLIW processor. It allows run-time switching of configurations, executing a large amount of contexts with low issue-width or a low amount of contexts with high issue-width.
This thesis investigates if the ρ-VEX processor can be competitively used as an accelerator using OpenCL. To answer this question, a design and implementation is made of such an accelerator. By measuring the speed of various components of this implementation, a model is created for the run-time of a kernel. Using this model, a projection is made of the execution time on an accelerator produced as an ASIC.
For the implementation of the accelerator, the ρ-VEX processor is instantiated on an FPGA and connected to the host using the PCI Express bus. A Linux kernel driver has been developed to provide interfaces for user space applications to communicate with the accelerator. These interfaces are used to implement a new device-layer for the pocl OpenCL framework.
By modeling the execution time, three major contributing factors to the execution time were found: the data transfer throughput, the kernel compile time, and the kernel execution time. It is projected that an accelerator based on the ρ-VEX processor, using similar production technologies and without architectural changes, can achieve 1.2 to 0.11 times the performance of a modern GPU.

CE-MS-2015-11

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Hugo van der Wijst
born in The Hague, The Netherlands

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

An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL

by Hugo van der Wijst

Abstract

In recent years the use of co-processors to accelerate specific tasks is becoming more common. To simplify the use of these accelerators in software, the OpenCL framework has been developed. This framework provides programs a cross-platform interface for using accelerators.

The ρ-VEX processor is a run-time reconfigurable VLIW processor. It allows run-time switching of configurations, executing a large amount of contexts with low issue-width or a low amount of contexts with high issue-width.

This thesis investigates if the ρ-VEX processor can be competitively used as an accelerator using OpenCL. To answer this question, a design and implementation is made of such an accelerator. By measuring the speed of various components of this implementation, a model is created for the run-time of a kernel. Using this model, a projection is made of the execution time on an accelerator produced as an ASIC.

For the implementation of the accelerator, the ρ-VEX processor is instantiated on an FPGA and connected to the host using the PCI Express bus. A Linux kernel driver has been developed to provide interfaces for user space applications to communicate with the accelerator. These interfaces are used to implement a new device-layer for the pocl OpenCL framework.

By modeling the execution time, three major contributing factors to the execution time were found: the data transfer throughput, the kernel compile time, and the kernel execution time. It is projected that an accelerator based on the ρ-VEX processor, using similar production technologies and without architectural changes, can achieve 1.2 to 0.11 times the performance of a modern GPU.

Laboratory : Computer Engineering
Codenumber : CE-MS-2015-11

Committee Members :

Advisor:	dr ir. Stephan Wong, CE, TU Delft
Chairperson:	dr ir. Stephan Wong, CE, TU Delft
Member:	dr. ir. Johan Pouwelse, PDS, TU Delft
Member:	dr. ir. Arjan van Genderen, CE, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

AHB Advanced High-performance Bus

AMBA Advanced Microcontroller Bus Architecture

API Application Programming Interface

ASIC application-specific integrated circuit

BAR Base Address Register

C2S Card to System

cdev character device

CPU central processing unit

DMA Direct Memory Access

FPGA field-programmable gate array

GPU graphics processing unit

HDL Hardware Description Language

IC instruction bundle count

ILP instruction-level parallelism

CPI cycles per instruction

IR Intermediate Representation

ISA Instruction Set Architecture

OpenCL Open Computing Language

PC Program Counter

PCI Peripheral Component Interconnect

PCIe PCI Express

pocl Portable Computing Language

RAM Random Access Memory

S2C System to Card

VEX VLIW Example

VLIW Very Long Instruction Word

XDMA Xilinx DMA

Acknowledgements

Developing this work has been a longer journey than I had anticipated. I would like to thank a number of people for their continued help and support during the last 14 months.

First of all I would like to thank my supervisor, Stephan Wong, for his guidance. Though few in number, our meetings forced me several times to adjust my goals, settling for more reasonable targets. I would also like to thank the PhD candidates Joost and Anthony, who were always available for a chat about the problem I was facing at that moment.

A lot of thanks goes to my fellow master students: Klaas, for helping me remember that I should take a break every few hours; Jens, for the opportunities to think and discuss different problems than the ones I was facing; and Jeroen, for rewriting the core, making all our lives easier.

A special thanks goes to Maarten, for going through the pain of proofreading a draft of this work. Our weekly lunches forced me to explain my research to somebody not intimate with the subject, while learning first hand how unclear such explanations are.

Finally, I would like to thank my parents, who are always interested in whatever I have to tell.

Hugo van der Wijst
Delft, The Netherlands
November 23, 2015

Introduction 1

This thesis describes the design and implementation of an OpenCL accelerator based on the ρ-VEX processor. This chapter provides an introduction to the problem. It starts with the context in which this work originated. Subsequently, the motivation for an accelerator based on the ρ-VEX processor will be discussed. This is followed by the problem statement that is researched in the thesis. The methodology to analyze the problem statement and the goals of the thesis are discussed. Finally, the outline of the rest of the work is given.

1.1 Context

Since the early 1990s, 3D graphics accelerators have become mainstream in common computers. These separate processors could perform 3D operations faster than the *central processing unit (CPU)*, allowing for richer 3D environments and more CPU time for other program logic. With the addition of programmable shaders, these *graphics processing units (GPUs)* became usable as more generic accelerators in the mid 2000s[1]. Manufacturers of GPUs embraced this new market and launched models specially designed to be used as accelerator.

The use of GPUs as accelerators illustrates the rise of the heterogeneous computing model, in which different types of processors are used in a system. In a typical heterogeneous computer there is one main program that divides the input into small work-items. Small programs called kernels need to be executed on each of these work-items, after which the work-items are combined into the output. Kernels are often run on only one type of processor, as the performance of a kernel differs for the different types of processor.

The increase in the use of heterogeneous computing has been especially visible in the High Performance Computing community. Today, many of the fastest super computers use two or more types of computing devices [2]. With the advent of CUDA and later OpenCL, accelerators are starting to be used in commercial applications[3][4].

In 2008 the Computer Engineering group of the TU Delft started working on a flexible and parameterized processor, the ρ-VEX[5]. This processor was designed to be used in a heterogeneous computing environment, as a coprocessor in a MOLEN[6] machine. Since its original implementation, the ρ-VEX has since been extended multiple times[7][8]. It currently supports run-time reconfiguration of the issue-width, balancing between running multiple threads with a low issue-width or less threads with a higher issue-width. A more extended discussion of the ρ-VEX can be found in Section 2.1.

1.2 Motivation

The reconfigurable nature of the ρ-VEX processor makes it an interesting architecture for an accelerator device. Accelerators based on GPUs are mainly optimized for vector calculations with a small amount of divergent branches. This use case can be performed relatively well by the ρ-VEX as it has a *Very Long Instruction Word (VLIW)* architecture, being able to execute up to 8 instructions per cycle.

GPUs perform poorly on kernel instances with a large amount of divergent branches. By reconfiguring the ρ-VEX processor as four 2-issue cores, a ρ-VEX-based accelerator can also perform well for these types of kernels.

Using the ρ-VEX as an accelerator gives rise to all kinds of new research questions, such as how to determine the best processor configuration based on a kernel and how to schedule different kernels. In order to enable this research, first an accelerator based on the ρ-VEX processor needs to be designed.

1.3 Problem statement, methodology, and goals

Currently on of the most popular cross-platform and cross-device heterogeneous computing framework is *Open Computing Language (OpenCL)*. This framework supports modern GPUs and CPUs, among other devices. Support for OpenCL is required for a quick adoption of a new accelerator. In this thesis, we investigate the performance that an accelerator based on the ρ-VEX processor can achieve using the OpenCL framework. The problem statement of this thesis is:

In the OpenCL environment, can the ρ-VEX be competitively used as an accelerator?

To answer this question, we first need to identify and measure the components that influence the performance of an accelerator. In order to perform these measurements, an accelerator platform using the ρ-VEX processor is made. This platform consists of an acceleration device, a communication mechanism between the acceleration device and the host, and a software library to interface with the acceleration device. Using the measured performance of the components, a model is made for the total execution time of a kernel. This model is subsequently validated.

To summarize, the main goals of this thesis are:

1. Designing and implementing a platform to use the ρ-VEX processor as an accelerator. Such a concept should consist of:
    - An acceleration device based on the ρ-VEX processor.
    - A communication mechanism between the acceleration device and the host.
    - An acceleration library to allow programs to use the accelerator.
2. Determining the performance of the different components of the acceleration platform.
3. Creating and validate a model of the total execution time of a kernel.

1.4 Organization

The rest of this thesis is organized as follows.

Chapter 2 provides background information on the technologies used in the rest of this thesis. It starts with an overview of the ρ-VEX project, describing both the hardware design and the available toolchain. Subsequently, the OpenCL framework is discussed, followed by a short description of the PCI Express bus.

Chapter 3 details the conceptual design of the system. First the proposed overall design is discussed. Next several different implementation for the OpenCL runtime are evaluated and an implementation is chosen. The additions necessary to support the accelerator in this runtime are discussed subsequently. This is followed by a presentation of the architecture of the accelerator device based on the ρ-VEX processor. Finally, the design of the software needed on the host to perform the communication with the accelerator is discussed.

Chapter 4 provides a description of the implementation. It starts with the implementation of the PCI Express communication on the accelerator. This consists of the DMA interface and the register interface. The Linux drivers that allow user space programs to communicate with the accelerator are discussed next. The chapter concludes with a description of the ρ-VEX back end of the OpenCL run time.

Chapter 5 explains the measurements performed to answer the research question. First the methodology and setup of the tests are detailed. Subsequently, the results for the communication throughput and latency, cache misses, compile time, and kernel execution time will be given. Based on these results a model for the full execution time will be proposed and validated. Finally, the available compilers will be benchmarked to explore the improvement potential in that area.

The found results are discussed in Chapter 6. It starts with a discussion of possible techniques to improve each of the factors identified in the model. This is followed by an analysis of the effect of improving these factors on the total speedup. Finally, a projection of the performance of the accelerator when implemented as an ASIC is made. These projected performance values are compared against both the reference platform and modern GPUs.

Concluding the thesis, Chapter 7 summarizes the previous chapters, lists the main contributions, and gives suggestions for future work.

2 Background

In the previous chapter we described the reason to design an accelerator based on the ρ-VEX processor. This chapter provides background information on the technologies used in the rest of the thesis.

First we give a short summary of the history of the ρ-VEX project, followed by an overview of the current design of the ρ-VEX. Subsequently we detail the available tool chain for the ρ-VEX, focusing mostly on the available compilers. We continue with an overview of OpenCL, a platform for heterogeneous computing, and finish with a short description of the PCI Express bus.

2.1 ρ-VEX processor

The ρ-VEX is a processor designed around the VEX *Instruction Set Architecture (ISA)*. VEX belongs to the category of *Very Long Instruction Word (VLIW)* ISAs. In a VLIW architecture, a processor executes multiple instructions in parallel. Which instructions are executed in parallel is decided during compilation by the compiler. The group of instructions that get executed in a cycle is called an (instruction) bundle.

Not all instructions can be executed in the same cycle. There are two reasons why instructions can not be scheduled in the same bundle: architectural restrictions and data dependencies. The architecture defines how many instructions of a certain type can be executed per cycle. Most architectures for example have fewer load-store units than arithmetic units, so only a limited amount of load or store instructions can be executed per bundle. Data dependencies exist between two instructions when the result of the first instruction is used by the second. Two instructions that have a data dependency can not be placed in the same cycle, as the result of the first is only available in the next cycle. The amount of instructions that can be executed per cycle is called the *instruction-level parallelism (ILP)*.

The design of *VLIW Example (VEX)* allows variation of multiple parameters without changing the ISA[9]. It was designed for educational purposes, allowing students to explore the design space of a VLIW ISA. The VEX ISA is based on the Lx ISA[10], which is used by the ST200 processor family from STmicroelectronics.

In the original ρ-VEX design by Van As, the following parameters could be changed at design time[5]:

- Issue-width
- Number of ALU units
- Number of MUL units
- Number of general purpose registers (up to 64)

- Number of branch registers (up to 8)
- Accessibility of functional units per lane
- Width of memory buses

Being able to change parameters at design time allows for optimizing the design of a processor for a specific application. While this is useful when only applications with a similar resource usage are run on the processor, often multiple applications with different usage patterns need to be executed. Having a fixed issue-width can cause programs with high ILP to run less quickly than is possible, while resources are not being used when program with low ILP are run.

To solve the problem of running programs with varying ILP efficiently, the ρ-VEX has been modified to support dynamically reconfiguring the issue-width of the core[7]. In this design the issue-width can be changed at run-time, switching between either multiple concurrently running contexts with a low issue-width or few contexts with high issue-width. The current version of the ρ-VEX supports up to four cores with issue-widths ranging from 2 to 8.

2.1.1 Hardware design

During development, the ρ-VEX processor is being instantiated on a *field-programmable gate array (FPGA)*. An FPGA is a chip that consists of many small programmable logic blocks that can be connected in different ways. A designer can specify the behavior of the chip, which is often done using a *Hardware Description Language (HDL)*. A synthesizer program turns this specification into an FPGA configuration file. Using this configuration file the FPGA can be reconfigured. The quick reconfiguration time and recent increases in clock speed and amount of available programmable logic blocks makes FPGAs highly suitable for the prototyping of logic designs.

The ρ-VEX processor currently supports instantiation on the Xilinx ML605 platform, which contains a Virtex-6 FPGA. There are two different configurations of the processor for this platform: a stand-alone version and a version using GRLIB. The stand-alone version does not have any external dependencies and implements its memory in block RAMs on the FPGA. The other version uses the GRLIB IP library[1] to give the ρ-VEX processor access to the DDR3 memory on the ML605.

The GRLIB library supports multiple FPGA boards and provides implementations of many different peripherals. It is developed by Cobham Gaisler AB and serves as a platform for the LEON3 processor core. The peripherals and processor cores are connected through the *Advanced High-performance Bus (AHB)*, a version of the *Advanced Microcontroller Bus Architecture (AMBA)* interconnect bus designed by ARM. Both commercial and GPL licensed versions of the library are available.

An overview of the ρ-VEX processor platform using the GRLIB library can be found in Figure 2.1. The ρ-VEX processor core is connected to the main GRLIB bus through a bus bridge that translates ρ-VEX bus requests and responses to AMBA bus requests and

[1] Intellectual Property library: In the context of an FPGA, an IP library often refers to a collection of HDL descriptions of components.

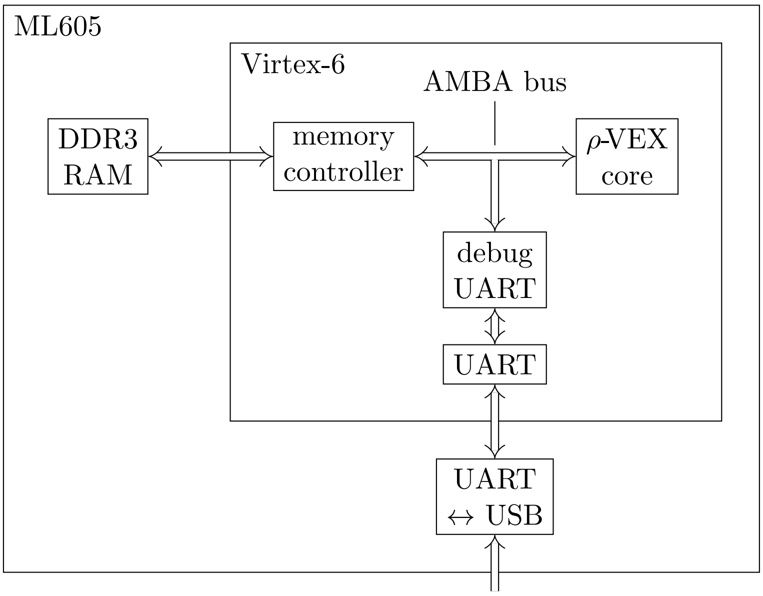

Figure 2.1: Overview of the GRLIB platform with ρ-VEX

responses. The memory controller is provided by the GRLIB platform. A UART debug block connected to the AMBA bus allows uploading and debugging programs running on the core. The UART debug block is part of the ρ-VEX project.

2.1.2 Tool chain

There exists a rich tool chain for the compilation and debugging of programs on the ρ-VEX processor. We will first provide an overview of the general compilation process of a program. Subsequently, the available compilers will be discussed, followed by a description of the other available tools.

The process to get from high level source code in a C-like language to machine code consists of several steps, as is depicted in Figure 2.2. The first step is compiling the source code files to assembly, which is a human readable version of machine code. Next, an assembler program translates the assembly file into an object file containing machine code and meta data. Since a program can consist of multiple source code files that reference methods or data declared in each other, a single object file might not be a complete representation of a program. Combining all the object files into a single executable is done by the linker.

In most C implementations some set-up work has to be performed before the `main` function is called. This work is performed by the `_start` routine, often located in a `_start.s` assembly file. On most platforms this file is provided by the linker.

Compiler A compiler translates source code files to assembly. This paragraph lists the different compilers that are available for the ρ-VEX processor.

The first compiler for the VEX ISA has been developed by Hewlett-Packard[11]. This proprietary compiler supports changing many different parameters of the target, and is

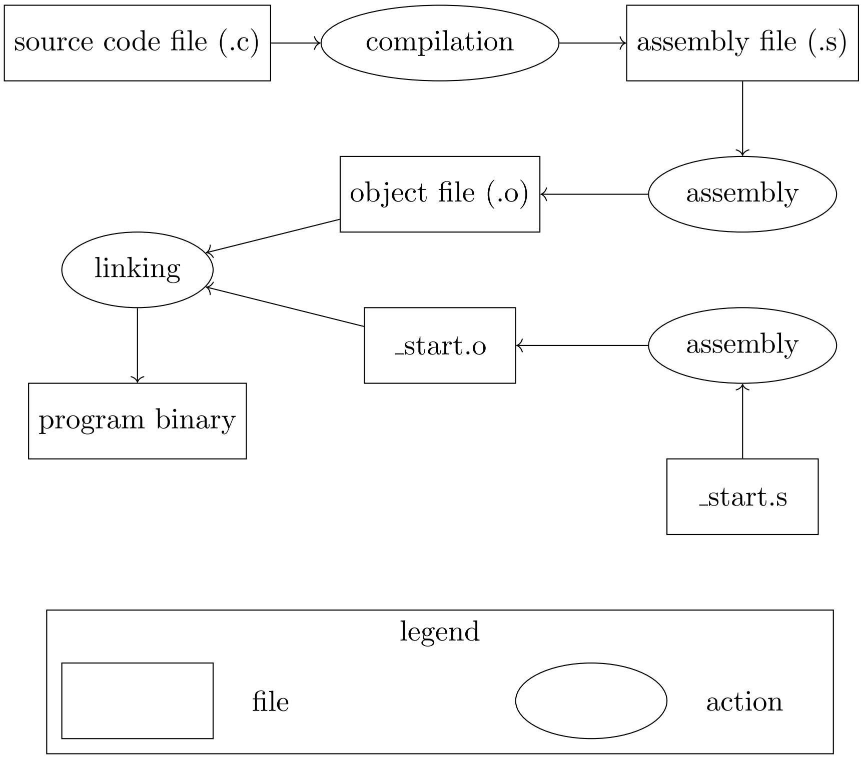

Figure 2.2: Overview of the steps performed to translate a program consisting of a single source code file into machine code.

used to analyze the design space offered by the VEX ISA.

The ST-200 microcontroller family by STMicroelectronics uses the Lx ISA. For this family, STmicroelectroncis developed an open-source compiler based on the Open64 compiler[12]. As the VEX ISA is based on Lx, a fork of that compiler has been made to target the ρ-VEX processor.

In 2012 IBM created a GCC back end for the VEX ISA, and in 2014 an LLVM back end has been created by Daverveldt[13]. Because of interest from the Computer Engineering group, a code generator for the CoSy compiler framework has been created. A detailed description of this code generator can be found in Appendix B.

Other tools In addition to a compiler, several other tools are needed to program and use the ρ-VEX. For programming, an assembler and linker complete the generation of machine code. For debugging, an ISA simulator and debugger for the ρ-VEX exist.

STmicroelectronics added support for the Lr architecture to the GNU assembler and linker, which are a part of the GNU binutils project. The GNU assembler and linker are the de facto standard tools used on most UNIX systems. The ρ-VEX assembler and linker are based on the GNU binutils work done by STmicroelectronics[12].

For debugging purposes, STmicroelectronics developed an ISA simulator for the Lr architecture. A modified version of that simulator called xstsim supports the VEX architecture.

```
__kernel void (__constante int *a, int b, __global int *c) {
  size_t i = get_global_id(0);
  c[i] = a[i] * b;
}
```

Figure 2.3: OpenCL kernel. This kernel performs an element-wise multiplication of a vector `a` and an integer `b`, storing the result in vector `c`. The function `get_global_id` returns the global ID of the current work-item.

Interfacing with in instantiated ρ-VEX core can be done using the debug interface. This interface consists of a debug server (`rvsrv`), a debug client (`rvd`), and a trace interface client (`rvtrace`). The debug server communicates over USB with the UART debug interface on the core. The debug client allows reading and writing from and to memory and registers, uploading programs and controlling the execution of the core. With the trace interface, jumps and writes to general purpose registers and memory can be logged to analyze execution. Communication between `rvsrv` and the clients is implemented over TCP/IP, allowing the server and the clients to run on different systems.

2.2 OpenCL

In this thesis an accelerator is designed that can be used with the *Open Computing Language (OpenCL)*. OpenCL[14] is a platform-independent framework for heterogeneous computing. It is based on kernels that are compiled at run-time for the accelerators available in the system. It has originally been developed by Apple in 2008, and has since been adopted by the majority of the industry. It is currently being developed by the Khronos Group.

OpenCL distinguishes between a host program and one or multiple kernels. The host program is executed by the user and runs on the *central processing unit (CPU)* of the computing device. The kernels are functions that execute on an OpenCL device. It is the responsibility of the host program to manage the execution of the kernels.

An OpenCL kernel is written in the OpenCL C programming language, which is a subset of the ISO C99 language with several extensions. See Figure 2.3 for an example of an OpenCL kernel. To allow the kernels to work on every device, they are distributed in source form with the program. During execution of the host program, the kernels are compiled for the specific device that they are requested to be run on.

The goal of OpenCL is to facilitate parallel programming. This is done by specifying an index space when queuing a kernel for execution. For each index in the index space, a kernel instance will be run. Such a kernel instance is called a work-item. The index space of the work-items can have up to three dimensions. All work-items are provided with the same arguments; the index of a work-item can be used to determine what work has to be performed.

Multiple work-items can be combined into a work-group, allowing for more coarse-grained control. The index space of the work-groups has the same dimension as the work-items. All of the work-items in a work-group are executed concurrently.

Figure 2.4 contains an example of a 2-dimensional index space. The space consists of

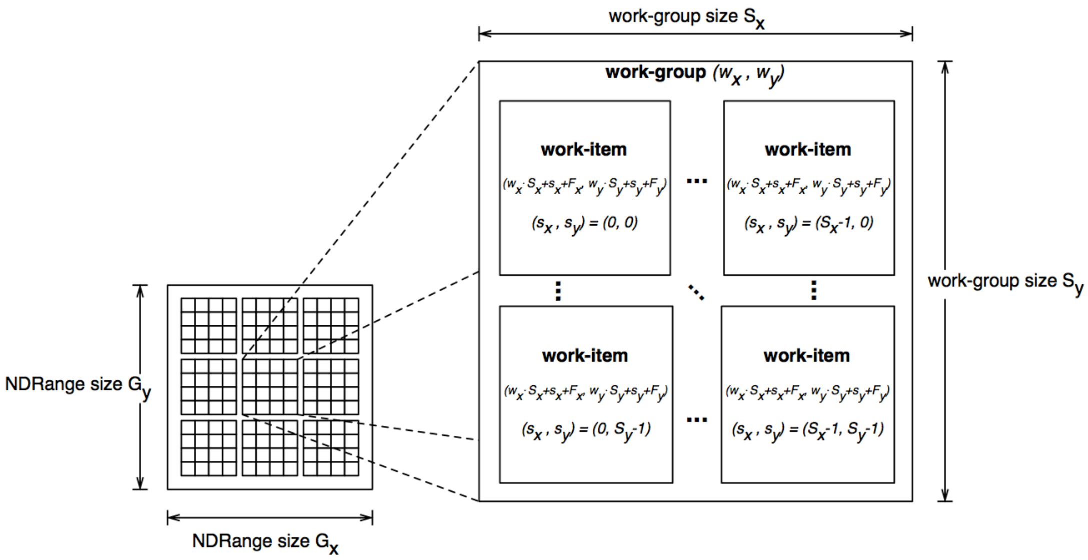

Figure 2.4: An example of a 2-dimensional index space (from [14])

G_x by G_y work-items. In this example there are nine work-groups, each containing S_x by S_y items. Each work-group has an ID, (w_x, w_y), and each work-item has both a local and a global ID. Global offsets F_x and F_y can be used to give an offset to the global ID of a work-item.

In the OpenCL C language there are four types of address spaces for variables: `private`, `global`, `constant`, and `local`. The private address space is used for all variables that can only be accessed by one work-item. Variables in the global address space can be accessed by all kernel instances, independent of the work-group they are in. This is the address space where buffers are located. The constant address space is for data that can be accessed by multiple different kernel instances, but is not writable. Finally, the local address space is shared between kernel instances within a work-group. Variables in the local address space are only allowed as arguments to a kernel or when declared in a kernel, but not in a non-kernel function.

A typical OpenCL program has to perform several steps to execute a kernel on an accelerator. It first needs to query the available devices and select the ones it wants to use. Subsequently, all OpenCL C language files required to build the kernel need to be loaded with the `clCreateProgramWithSource` function. The program should then be build by calling the `clBuildProgram` function.

Actions on a device are stored in a command queue, which is created by the `clCreateCommandQueue` function. The `clEnqueueWriteBuffer` function will upload buffers to the accelerator. To execute a range of kernel instances, the `clEnqueueNDRangeKernel` function is used. Finally, results are read using the `clEnqueueReadBuffer` function.

There are several other heterogeneous programming specifications such as CUDA, OpenACC and OpenHMPP. Most of these specifications target a specific device at compile time (OpenACC, OpenHMPP). While CUDA targets multiple devices, these are all NVIDIA graphics cards. As the CUDA framework is not open, it is not possible to add support for other devices.

2.3 PCI Express

The accelerator created in this thesis uses the *PCI Express (PCIe)* bus to communicate with the host. PCIe is an interconnect used in many modern PCs and servers. It is the successor of the *Peripheral Component Interconnect (PCI)* bus, and shares many of its concepts. In this section a brief overview of the PCIe design will be given. For a more extensive introduction, see [15].

There are currently three versions of the PCIe bus. All versions are backward and forward compatible with devices made for other versions. A device can have between 1 and 16 lanes, each lane allowing for simultaneous transmission and retrieval of one byte stream. With PCIe version 1 the theoretical bandwidth is 250 MB/s per lane per direction. This bandwidth has increased to 984.6 MB/s per lane per direction in version 3 [16].

A PCIe bus consists of a network of switches and devices. A device can have different functions, that can be addressed separately. Every device function can initiate a transfer to every other device function. Transfers consist of one or more packets, routed by the switches to the receiver. Different classes of packets can be assigned, allowing for traffic shaping.

An example PCIe topology can be seen in Figure 2.5. The PCIe root complex connects the CPU and memory of the system to the PCIe bus. This allows CPU memory requests to be handled by a device on the PCIe bus. It also allows devices access to the system memory without involving the CPU. This principle is called *Direct Memory Access (DMA)*.

From the software perspective, the PCIe bus is an extension of the PCI bus. It recognizes four types of communication: configuration space requests, memory requests, I/O requests and messages. Every device function has a configuration space of 256 to 4096 bytes (256 bytes in PCI). Registers in the configuration space are 4-bytes and can be read or written one at a time. Memory requests allow variable length reads and writes, up to 4096 bytes. I/O requests are meant for reads and writes to I/O registers. The size of these requests is limited to 4-byte reads or writes. Messages are used for several different functions, such as interrupts and power management.

The configuration space of a PCIe device contains the device functions and multiple configuration parameters for each function. One of these parameters is the size of a region in the host memory space that is mapped to the device function. The location of this region is written to the *Base Address Register (BAR)*. Any read or write requests to that memory region will be turned into memory requests and send to the correct device.

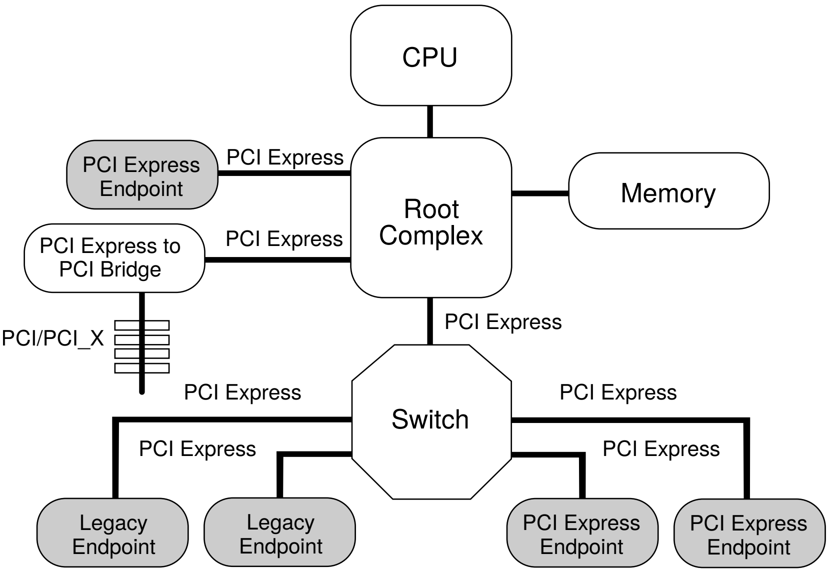

Figure 2.5: Example PCI Express topology (from [15])

2.4 Conclusion

This chapter presents the different technologies necessary to understand the rest of the thesis. It starts with an overview of the ρ-VEX processor, which is a processor with a run-time reconfigurable issue-width and core count. ρ-VEX processors are currently instantiated as a soft-core on an FPGA. The GRLIB IP library is used to provide access to DDR3 *Random Access Memory (RAM)* on the Xilinx ML605 platform. A rich tool chain is available for the ρ-VEX processor. It consists of multiple compilers, a port of the GNU binutils project, and debugging tools.

As the created accelerator uses the OpenCL framework, a short introduction to the OpenCL framework is given. This technology defines a framework for programs to execute kernels on multiple compute units. These kernels are shipped as source code and compiled by the framework implementation for the correct architecture. Several other compute related technologies are also mentioned.

Finally, a short overview of the functioning of the PCIe bus is given. This bus provides a high-speed interconnect between peripherals and the CPU. It allows mapping parts of the system memory range to a device for direct access by the CPU. Additionally, devices can perform DMA by accessing system memory without involving the CPU.

Concept 3

This chapter details the conceptual choices for the OpenCL implementation that has been made. First the overall system design is discussed. Subsequently, several different implementation for the OpenCL runtime are evaluated and an implementation is chosen. The additions necessary to support the accelerator in this runtime are discussed next. This is followed by a presentation of the architecture of the accelerator device based on the ρ-VEX processor. Finally, the design of the software needed on the host to perform the communication with the accelerator is discussed.

3.1 System design

An overview of the system in which the ρ-VEX is going to be integrated can be found in Figure 3.1. The system consists of a *central processing unit (CPU)*, main memory, a southbridge and a *PCI Express (PCIe)* bus, which are connected through the northbridge. The PCIe bus connects multiple different devices with the CPU and the main memory. The accelerator with ρ-VEX processor will be instantiated on the Virtex-6 *field-programmable gate array (FPGA)*. This FPGA is located on the ML605 platform, which is connected to the PCIe bus.

The PCIe bus and the main memory are both connected to the northbridge. This allows PCIe memory requests to access the memory without involving the CPU, enabling *Direct Memory Access (DMA)*. Additionally, the CPU can directly communicate with a PCIe device function by performing memory read and writes to the memory at the *Base Address Register (BAR)*. These accesses will be translated by the northbridge and sent as memory request packets to the correct device.

There are two processors in this system that can be controlled by the user: the CPU

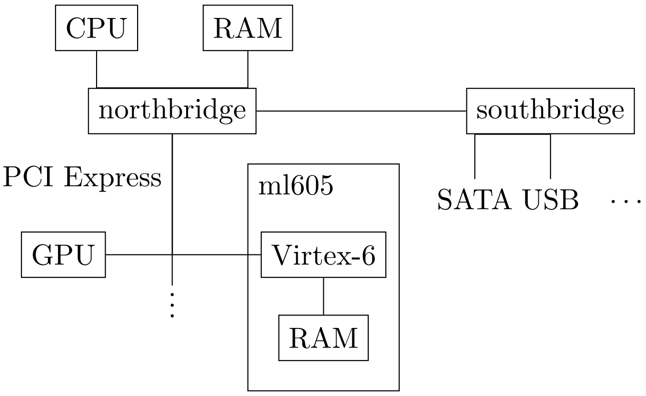

Figure 3.1: System overview

Project	Supported accelerator devices
Beignet[18]	Intel GPUs
Clover[19]	AMD and NVIDIA GPUs
FreeOCL[20]	Host CPU
pocl[21]	Host CPU, extendible

Table 3.1: Overview of open-source OpenCL implementations and their supported accelerator devices

and the ρ-VEX soft core located on the Virtex-6 FPGA. Consequently, there are two locations code can be executed. In the remainder of this thesis, the CPU will be referred to as *the host*, while the accelerator is often referred to as *the device*.

3.2 OpenCL runtime

An OpenCL implementation consists of several components. First of all there are two *Application Programming Interfaces (APIs)*: the OpenCL runtime API and the OpenCL C language API. The runtime API is the interface that the program uses to compile, load and run kernels and buffers on the device. The kernels are written in OpenCL C, which has several built-in functions. These built-in functions make up the OpenCL C API.

As the kernels need to be build locally, an OpenCL C compiler is needed. Depending on the device, other parts of the compiler tool chain such as assemblers and linkers might be needed. The final part of an OpenCL implementation is a communication system with the device. This system should be able to initialize, start and stop the device.

There are many existing OpenCL implementations, both open-source and proprietary. Because the OpenCL specification is quite large and to avoid duplicating effort, it was decided to not start a new implementation from scratch.

For an overview of the features of the OpenCL framework, see Stone, Gohora and Shi [17], for the precise definition see the specification itself[14].

3.2.1 Available implementations

There are several different OpenCL implementations available. These implementations can be divided in open source and proprietary implementations. As the goal is to add a new target to the implementation, only open source implementations are considered.

There are currently four open source OpenCL implementations: Beignet[18], Clover[19], FreeOCL[20] and *Portable Computing Language (pocl)*[21]. An overview of the accelerator devices that these implementations support can be found in Table 3.1.

As a basis for the ρ-VEX OpenCL back end, using Beignet and Clover would require a lot of work as they are tightly coupled with the rest of the Linux graphics stack. This leaves two implementations that could be extended: FreeOCL and pocl. FreeOCL aims to support multiple different compilers while supporting only one accelerator device: the host CPU. pocl is designed for use with one compiler framework but with multiple

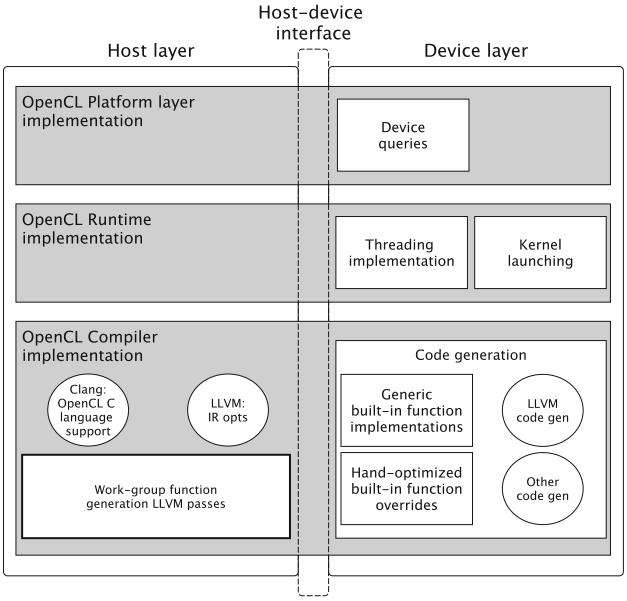
Host-device interface
Host layer
Device layer
OpenCL Platform layer implementation
Device queries
OpenCL Runtime implementation
Threading implementation
Kernel launching
OpenCL Compiler implementation
Clang: OpenCL C language support
LLVM: IR opts
Work-group function generation LLVM passes
Code generation
Generic built-in function implementations
LLVM code gen
Hand-optimized built-in function overrides
Other code gen

Figure 3.2: Subcomponents of the pocl OpenCL implementation (from [22])

different devices as accelerators. As we want to add a new accelerator, we chose to extend the pocl implementation.

3.2.2 pocl extensions

The pocl OpenCL implementation is divided into a host layer and a device layer[22], see Figure 3.2. The host layer encompasses all the device independent functions required for an OpenCL implementation. This includes an OpenCL C language compiler and generic optimization passes. The device layer is contains all device specific functionality, and can be seen as a hardware abstraction layer.

pocl uses the Clang compiler front end to compile the OpenCL C language files. This compiler generates LLVM *Intermediate Representation (IR)* instead of machine code. This IR is architecture-independent, allowing reuse of compiler front ends and optimization passes. Several back ends are available to translate the IR to architecture specific assembly.

The device layer contains several components that need to be implemented to add a new device target. To determine how many available devices the OpenCL implementation can access, the layer contains a device query component. The device layer is also responsible for the generation of the machine code to execute on the device. This machine code should be generated from the IR that the host layer delivers, combined with the built-in OpenCL C language functions. Finally, the device layer has to be able to manage the device, implementing data transfer, setup and execution control.

As pocl uses LLVM, the ρ-VEX back end for this compiler developed by Daverveldt [13] is used to compile from the LLVM IR to VEX assembly. For the other tools needed to translate the assembly to machine code, the programs from the binutils project are used.

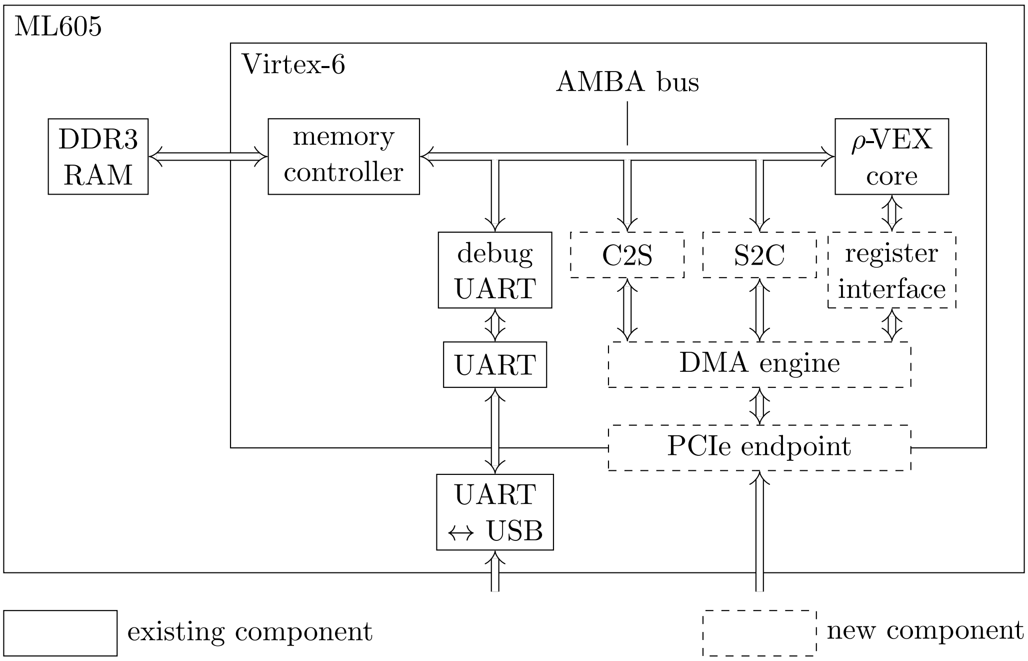

Figure 3.3: Overview of the additions to the platform

Faddegon[23] and ACE have worked on a bridge between LLVM IR and CoSy's IR: LLVM-TURBO. This new technology allows passes and code generators developed for the CoSy compiler framework to be used with LLVM front end tools. A development version of this technology will be used with the created CoSy code generator (see Appendix B) to evaluate its performance with pocl.

3.3 Accelerator platform

The accelerator is built on the Xilinx ML605 platform. The ρ-VEX processor is instantiated as a soft core on the Xilinx Virtex-6 FPGA located on that platform. Because a large amount of data will need to be stored on the accelerator, access to the *Random Access Memory (RAM)* on the ML605 is necessary. Therefore, the GRLIB configuration of the ρ-VEX processor will be extended. To allow communication between the host and the accelerator, the PCIe bus is used. The ML605 is designed as a PCIe extension card and has 512MB DDR3 RAM. The PCIe endpoint support up to 8 lanes for PCIe version 1 and 4 lanes for PCIe version 2.

The necessary additions to the ρ-VEX GRLIB platform can be seen in Figure 3.3. For the DMA engine an IP core from Northwest Logic is used[24]. This core is retrieved from the Virtex-6 FPGA Connectivity Targeted Reference Design[25].

There are three interfaces that this DMA core exposes: a *Card to System (C2S)* interface, a *System to Card (S2C)* interface and a register interface. The S2C and C2S interfaces are used for DMA transfers to and from the card respectively. The register interface is used to implement application specific registers accessible using memory

name	type	description
run	input	Enables or disables execution of a context.
reset	input	To reset the state of a context.
resetVector	input	The value that the *Program Counter (PC)* should be set to during a reset.
irq	input	Request an external interrupt to be triggered.
irqID	input	The interrupt argument of the triggered interrupt.
done	output	Set high if the core has executed the STOP instruction.
idle	output	High if the core is currently executing. The core can be running and idle when it is done.
irqAck	output	High when a requested interrupt is triggered.

Table 3.2: Run control signals

requests.

The S2C and C2S interfaces will be connected to the *Advanced Microcontroller Bus Architecture (AMBA)* bus. This bus also connects to the RAM and the ρ-VEX core. DMA transactions can thus be used both to read and write to the memory and to the ρ-VEX configuration registers. As DMA transfers are performed by the C2S and S2C engines, they can be executed independently from programs running on the ρ-VEX core. All DMA transactions will be initiated by the host; no interface is made available for the device to start a transfer.

The register interface will be connected to the run control interface of the core. This run control interface consists of several signals per context, these are listed in Table 3.2. Accesses to the register interface have no latency as no bus requests are necessary to handle them.

3.4 Host PCIe interface

For the OpenCL program to communicate with the accelerator, it needs access to the PCIe bus. As it should not be required to run an OpenCL program in elevated mode, a separate component needs to handle the communication with the accelerator. This is achieved by creating a Linux driver that handles the communication between the PCIe subsystem and the pocl library.

An overview of the host software stack can be found in Figure 3.4. The program communicates with the host layer of pocl. The host layer uses the external Clang component to compile kernels to LLVM IR, and delegates device specific requests to the ρ-VEX device layer. This device layer will use the VEX LLVM back end to translate the IR representation of a kernel to assembly. This assembly will be assembled and linked using the binutils programs.

Communication with the ρ-VEX core is handled by two Linux kernel drivers. The Xilinx DMA (`xdma`) driver performs the initialization of the PCIe subsystem in the Linux

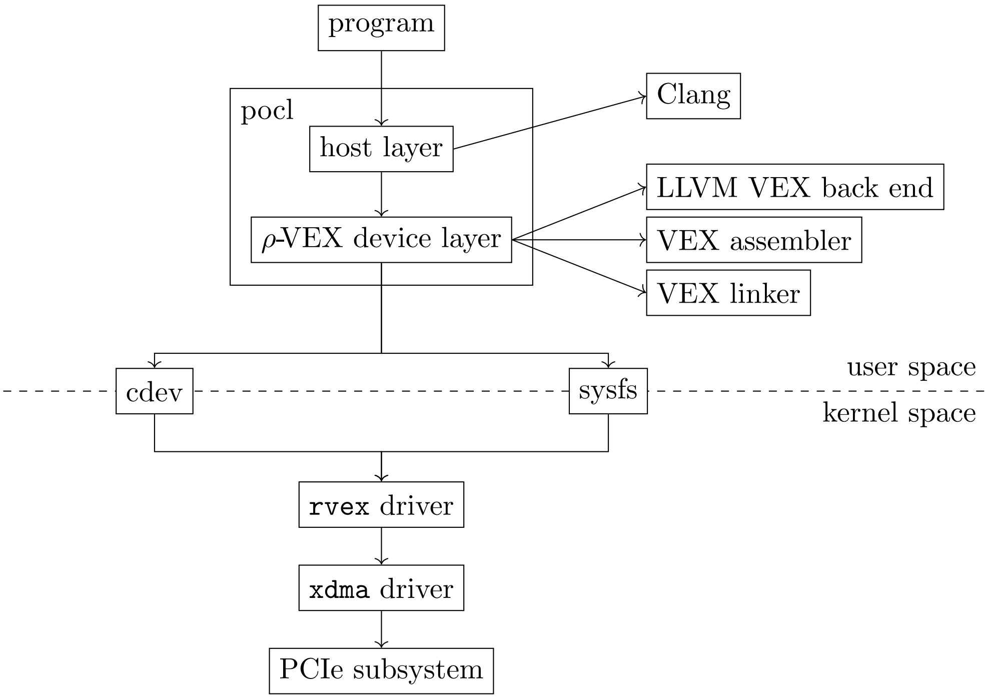

Figure 3.4: Overview of software on the host. Arrows point from the component initiating communication to the component that responds.

kernel and handles DMA transfers to and from the Northwest DMA engine. The `rvex` Linux driver provides a convenient interface for user space programs, allowing transfers to memory and register accesses. It uses the `xdma` driver to communicate with the core.

The `rvex` Linux driver provides two different interfaces to user space: a *character device (cdev)* interface and a sysfs interface. The character device gives user space programs a direct way to access the memory space on the board. It creates a file in the `/dev/` directory. Reads and writes to this file are handled by performing DMA transactions to the ML605.

The sysfs interface is used to access core and context registers and the run control signals. Each register and signal will be represented as a separate file in a subdirectory located in the `/sys` directory. A read or write to such a file will be handled by either a performing a request to the DMA register interface or by performing a DMA transaction.

3.5 Conclusion

This chapter presents the concept for the OpenCL accelerator implementation. It starts by giving an overview of the entire system, followed by descriptions of the components: the OpenCL runtime, the accelerator platform, and the communication interface.

As OpenCL runtime the open-source implementation pocl is chosen, as it is specifically designed to be used with architecturally different accelerators. To add support for the ρ-VEX based accelerator, a new device-layer is added to pocl. To compile the

LLVM IR to VEX assembly, the LLVM back end developed by Daverveldt will be used. Additionally, an evaluation version of the LLVM-TURBO technology will be evaluated for use with pocl.

The Xilinx ML605 platform will be used for the accelerator. The ρ-VEX processor will use the GRLIB configuration, as it provides a memory controller. To implement DMA transactions an IP core by Northwest Logic is used. This DMA interface gives the PC access to the bus connecting the core to the memory and other peripherals. A second interface using PCIe registers is available to control and monitor the execution of contexts on the core.

On the host the Xilinx DMA driver is used to handle DMA transactions. This driver is used by a new driver (`rvex`) that provides user space applications an interface to interact with the core. Access to the accelerator memory is given using a character device, while registers and execution control signals are available over the sysfs interface.

Implementation 4

In the previous chapter the conceptual design of the accelerator was discussed. This chapter describes its implementation. First, the implementation of the PCIe communication on the accelerator is described. This consists of the DMA interface and the register interface. The Linux drivers that allow user space programs to communicate with the accelerator are discussed next. Finally, the ρ-VEX back end of the OpenCL runtime is detailed.

4.1 Accelerator implementation

The *PCI Express (PCIe)* interface of the accelerator consists of a PCIe endpoint, a *Direct Memory Access (DMA)* engine, a *Card to System (C2S)* and *System to Card (S2C)* interface, and a register interface (see Figure 3.3). The C2S and S2C interfaces connect the DMA engine with the *Advanced Microcontroller Bus Architecture (AMBA)* bus, allowing DMA transfers to memory. The register interface implements several device specific registers for controlling the ρ-VEX core and specifying the design configuration.

This section starts with the chosen configuration of the PCIe endpoint and DMA engine. Subsequently, the implementation of the C2S and S2C interfaces are discussed. The section finishes with a description of the register interface.

4.1.1 PCIe endpoint

The PCIe endpoint is generated by the Xilinx CORE Generator System. The supported PCIe configurations of the Virtex-6 *field-programmable gate array (FPGA)* can be found in Table 4.1. During implementation it was found that routing a design with any configuration other than the 4-lane version 1 configuration often fails.

The theoretical throughput of a 4-lane version 1 configuration is 1 GB/s in both transmit and receive direction. This is 7 times slower than the maximum throughput of the AMBA bus connecting the DMA interface with the memory, which is 143 MB/s. No problems are therefore expected from using the 4-lane version 1 PCIe configuration.

Version	Lanes	Bandwidth (in each direction)
1	4	1 GB/s
1	8	2 GB/s
2	4	2 GB/s

Table 4.1: Supported PCIe configurations by Virtex-6

4.1.2 DMA engine

The DMA engine from Northwest Logic supports both addressed and streaming data transfers. As is implied by the name, an addressed transfer indicates the address on the device where data should be written to or read from. When the device consumes or produces a stream of data, addresses are not needed and streaming data transfers can be used.

For streaming transfers, the device waits until data is available from the host. Received data is processed and if a result is available, it is sent to the host. In this use case, the host determines when data is send to the device, and the device determines when it sends data to the host.

Addressed data transfers are designed as a separate layer on top of the streaming data transfers. When performing addressed transfers, the host controls when and what data is transfered. To initiate a transfer, the host sends an "addressed packet" containing the type, the start address, and length of the request. For writes the host appends the data that needs to be written, while for reads it expects the device to send the requested data.

As accesses on the AMBA bus are addressed, the DMA engine was configured to use addressed transfers.

4.1.3 C2S and S2C interfaces

The C2S and S2C interfaces connect the AMBA bus with the DMA engine. The C2S interface handles read requests from the host while the S2C handles write requests.

The C2S and S2C components are connected to the AMBA bus using ρ-VEX buses. This is done as the ρ-VEX bus is less complex and provides the necessary functionality. The bus bridge between the ρ-VEX bus and the AMBA bus is an existing component in the ρ-VEX project.

The DMA engine runs at 125 MHz. As the rest of the core runs at 37.5 MHz, the requested or provided data has to cross clock domains. This is done by a clock domain crossing for the ρ-VEX bus, which is also a part of the ρ-VEX project. For both the C2S and S2C components, a crossing is placed between the component and the bus bridge to the AMBA bus.

The C2S and S2C interfaces from the DMA engine transfer 64-bit words per request. Because the ρ-VEX bus has a width of 32-bits, every DMA request requires two ρ-VEX bus accesses.

For a detailed overview of the interface the DMA engine, see the manual distributed with the Xilinx ML605 reference design[25].

C2S Figure 4.1 shows the state diagram of the C2S component. The component starts in the `wait_pkt` state, where it waits for an addressed packet. This addressed packet specifies the location of the data and the amount of bytes to read.

When an addressed packet is available, as indicated by the `apkt_req` signal, the component goes into the `read_low` state. In this state the first 32 bits are requested over the ρ-VEX bus. When the bus acknowledges the request and provides the requested

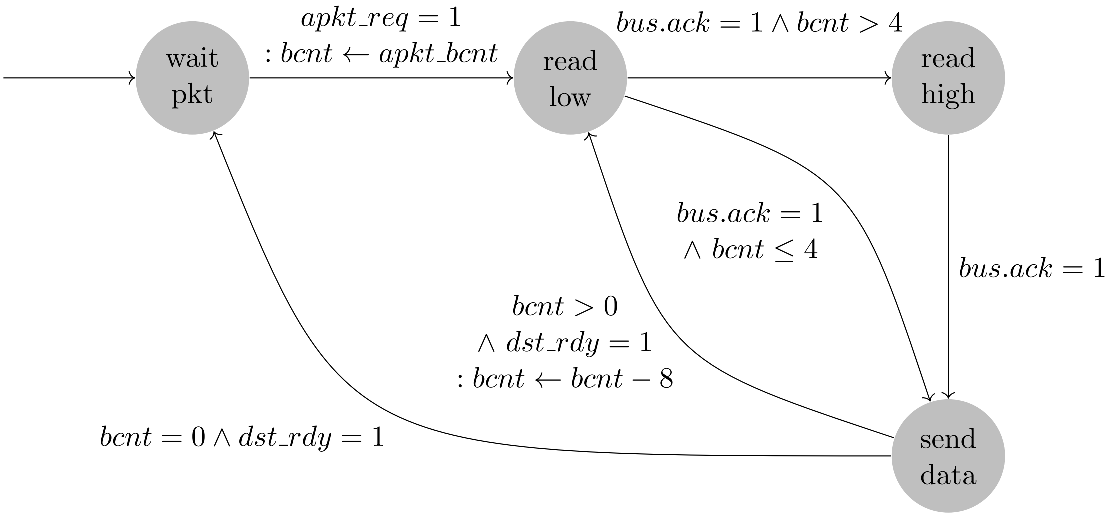

Figure 4.1: State diagram of C2S component. State transitions only occur on the rising edge of the clock.

data, the component moves to the next state. This next state is either the `read_high` state if more than 4 bytes needed to be read, or otherwise the `send_data` state.

In the `read_high` state the C2S component reads the second 32-bit word. When the data is received over the ρ-VEX bus, the component moves to the `send_data` state.

In the `send_data` state the data read in the previous states is provided to the DMA engine. After the DMA engine acknowledges the receipt using the `dst_rdy` signal, the component moves to the next state. This is either the `read_low` state if there are more bytes to read in this transaction, or the `wait_pkt` state otherwise.

An overview of the interaction between the different components is shown in a sequence diagram in Figure 4.2. The DMA engine initiates the transfer by providing an `addressed packet`. The C2S component then starts to perform read requests and provides the DMA engine with data packets to transmit. With the last data packet, the end-of-packet (`eop`) flag is set to one, indicating to the DMA engine that the packet is complete.

S2C A state diagram for the S2C component can be found in Figure 4.3. This diagram resembles the state diagram for the C2S component (Figure 4.1), but there are some differences. As the S2C component is used for writing data to the local bus, it has to wait for that data to be available from the DMA engine. When the data is available, it is written most significant byte first in up to two writes.

A difference between the S2C component compared to the C2S component is that not all writes are allowed. The memory on the ML605 requires writes to be aligned, and disallows 3-byte writes. To guard against bus faults due to these illegal memory accesses, such write requests are ignored. The host driver is expected to transform unaligned write requests into multiple aligned requests.

The interaction of the S2C component is shown in Figure 4.4. In contrast to the

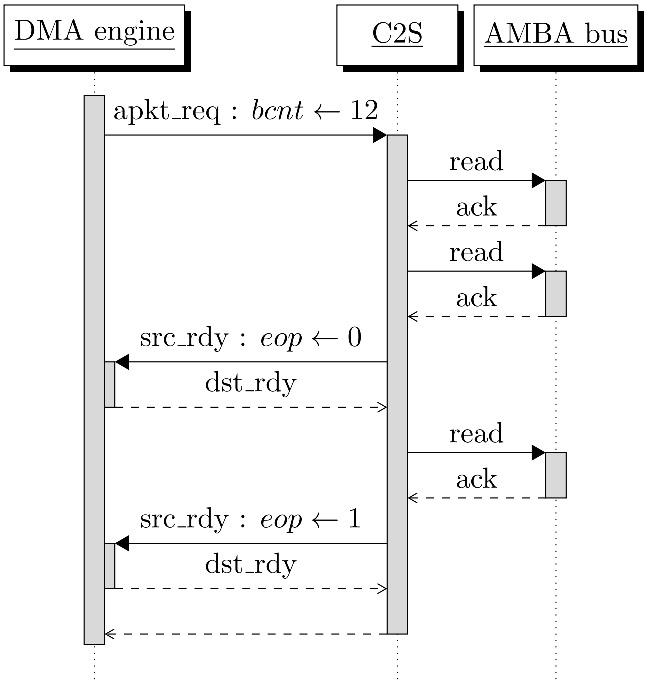

Figure 4.2: Sequence diagram of a read request.

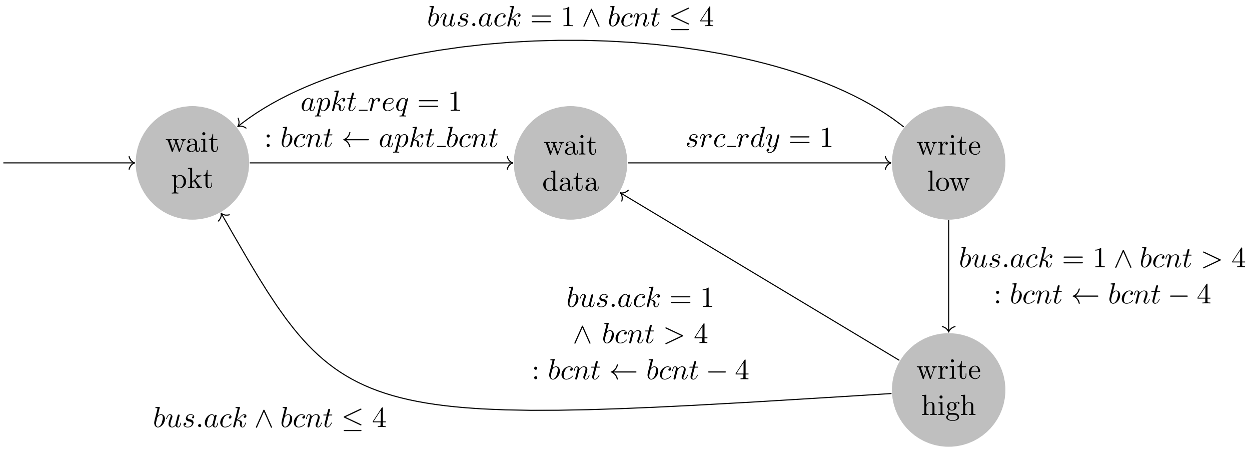

Figure 4.3: State diagram of the S2C component. State transitions only occur on the rising edge of the clock.

C2S component, the S2C component only performs actions when data is provided by the DMA engine.

4.1.4 Register interface

The register interface allows quick configuration and status retrieval of the device. The DMA engine implements a 64 kilobyte *Base Address Register (BAR)*. Accesses to the

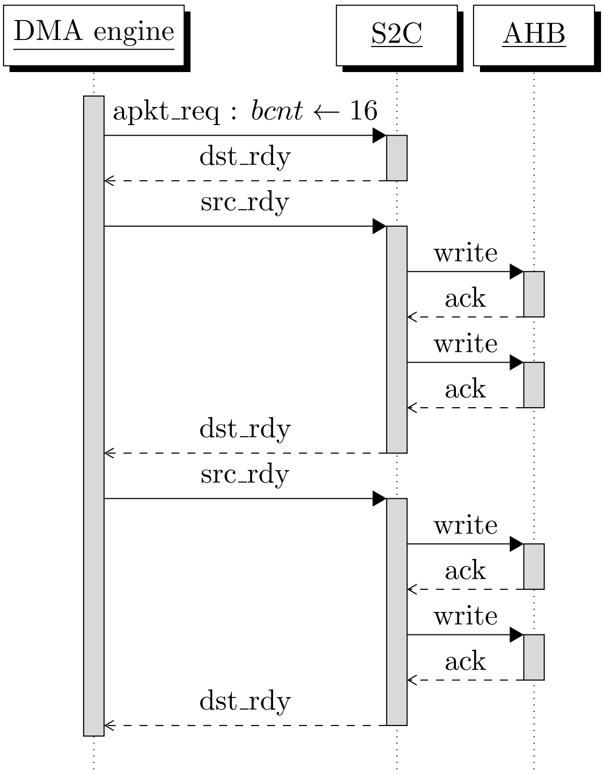

Figure 4.4: Sequence diagram of a write request.

registers are performed by memory read and write requests to that BAR range.

The first 32 kilobytes of the BAR are used by the DMA engine for registers to control the working of the DMA. The next 32 kilobytes are available for device specific registers and are used to control the ρ-VEX.

For the ρ-VEX eight device registers have been implemented (Table 4.2). Three registers are constant and indicate the version of the interface, the amount of cores on the device, and the amount of contexts in a core. The other 5 registers are used for the run control signals as described in Table 3.2.

The `run`, `idle`, `done`, and `reset` registers are 64 bits long. Each bit in the register encodes the signal for the respective context: bit 0 for context 0, bit 1 for context 1, etc. Every context has its own `reset_vector` register, which indicates the value of the *Program Counter (PC)* after a reset. This register is a 64 bits long, as that leads to a simpler implementation. As the address space of the ρ-VEX processor is 32 bits, the upper 32 bits of the `reset_vector` register are ignored.

4.2 Host interface

The interface between user space programs on the host and the accelerator PCIe interface is implemented by the `xdma` and `rvex` kernel driver (see Figure 3.4 in Chapter 3). The `xdma` driver initializes the PCIe subsystem of the Linux kernel, configures the DMA

address	register	size	access
0x8000	`iface_version`	64	R
0x8008	`no_contexts`	32	R
0x800C	`no_cores`	32	R
0x9000	`run`	64	R/W
0x9008	`idle`	64	R
0x9010	`done`	64	R
0x9018	`reset`	64	R/W
0x9200 - 0x93F8	`reset_vector`	64	R/W

Table 4.2: Device specific registers

engine, and handles DMA transfers. The `rvex` driver creates two interfaces for user space programs to communicate with core: a sysfs interface and a character device interface. It translates accelerator specific requests into `xdma` function calls.

4.2.1 `xdma` kernel driver

The `xdma` driver is developed by Xilinx and is distributed as part of the Virtex-6 Targeted Reference Design[25]. This driver was modified to make it 64-bit compatible, to allow addressed packets, and to allow a read DMA transfer.

In the supplied `xdma` driver the assumption was used that a host pointer fits in a 32-bit pointer. As the virtual address range the kernel allocates memory in uses the entire available address range, addresses allocated by a 64-bit kernel often don't fit in a 32-bit pointer. Casting such a 64-bit pointer to a 32-bit value and back to a pointer will result in a different value. While the driver would compile for a 64-bit machine, running it on such a machine immediately crashes the operating system as incorrect memory values are being used. This was fixed by using the correct platform-independent types for pointer values. As the DMA engine assumes that pointers are 32-bit, the physical addresses of allocated memory are forced to the 32-bit range with the `pci_set_dma_mask` function.

The reference design with which the `xdma` engine was supplied uses a streaming application. Because this does not require support for addressed packets, supplying an address for the card to indicate where to store the data was not implemented. To add addressed packet support, an additional DMA register had to be populated with the requested card address.

Another consequence of the design for streaming purposes was the lack of a host initiated DMA read transfer. In the streaming application, the host initiates DMA writes, while the device initiates the transfer back to the host when it is done processing data.

The ρ-VEX OpenCL back end needs to be able to initiate read requests from the accelerator memory, so this feature was added to the `xdma` engine. To allow the targeted reference design applications to continue functioning, the `xdma` driver can be used in both a streaming and an addressed mode. In the streaming mode the card initiates transfers from the card to the system, and in the addressed mode the hosts initiates these transfers.

4.2.2 `rvex` kernel driver

User space applications communicating with the accelerator need to perform two types of read and write requests: to memory and to registers. Requests to memory can start at any address in the address space and have a length determined by the user space application. Conversely, requests to a specific register will always be to the same address and of the same length.

Two different interfaces are used for these different types of requests. Memory transfers are handled by a character device interface, while register transfers are handle by a sysfs interface.

Character device interface A *character device (cdev)* in the Linux kernel is a device on which single byte reads and writes are possible. Such a character device is an abstracted view of a device, and does not necessarily represent all the functionality of a physical device.

A character device consists of a special file in the `/dev` directory. In case of the `rvex` driver this file is called `fpga0`. Standard UNIX file operations like `open`, `close`, `read`, `write`, and `seek` are implemented for the device. The `open` and `close` operations check if the user is allowed access to the device and respectively initialize or clean up necessary resources. The `seek` operation is used to set the memory location where the `read` and `write` operations should start.

Only the `read` and `write` operations need to communicate with the board. To implement these operations, DMA transfers are initiated. The calling process is blocked waiting for the transfer to complete.

DMA transfers read or write directly to the AMBA bus on the FPGA. This makes both the ML605 RAM and bus accessible ρ-VEX registers available through the cdev interface. It does require the user space application to know the memory layout on the platform.

sysfs interface The sysfs interface provides easy access to the PCIe registers of the accelerator. This interface is needed as these registers are not accessible through the AMBA bus. To provide more convenient access, the sysfs interface is also used to access the bus accessible ρ-VEX registers.

sysfs is a virtual filesystem in Linux mounted at `/sys`. It offers a method for kernel code to communicate different types of information to user space[26]. This is done by mapping kernel objects, attributes of these objects, and relationships between objects to respectively directories, files and symbolic links (see Table 4.3). User space programs can discover the devices and attributes by walking the directory tree. Reading attributes is done by reading the associated file; writing to that file will set the attribute when supported.

The `rvex` driver creates 3 types of devices with different attributes to model the accelerator: `fpga` devices, `core` devices and `context` devices.

kernel element	sysfs representation
kernel object	device
object attribute	file
object relationship	link

Table 4.3: Representation of kernel elements in sysfs filesystem

An accelerator is represented by a `fpga` device. As the memory is shared with all cores on the accelerator, the `fpga` device contains a link to the character device. The `fpga` device can have one or multiple `core` devices, that each can contain several `context` devices.

When the `rvex` driver gets started, it reads the `no_cores` and `no_contexts` PCIe registers to determine the amount of devices to create. These registers are not accessible to user space programs, but the information can be deduced by counting the amount of created devices.

The run control registers are attributes of the `context` devices. Reading a file associated with a binary register will give a 0 or 1 depending on the status of the register. If the register is writable, writing a 0 to the file will clear the correct bit in the register and writing a non-zero integer will set that bit. The `reset_vector` file reflects the 32-bit value of the reset vector register associated with the context the file is in.

The ρ-VEX core exposes multiple registers to control the way it functions. Accessing these registers from a user space application can be necessary to, for example, measure the performance of executed programs. As these registers are accessible over the AMBA bus, performing a DMA transfer to the correct address will read or write the register. This can be achieved by a user space program through the character device interface, though that requires knowledge of the size and memory locations of the registers.

To provide an easy interface to these registers in the ρ-VEX core, analogous attributes have been added to the `core` and `context` devices. A read or write command to the sysfs file will be fulfilled by a DMA transaction to the correct memory address.

Access to the general purpose registers of a ρ-VEX context is implemented in a special way. When reading the `gpregs` file of a context, the values of all 64 registers are returned separated by spaces. When writing a series of space separated values is expected. The first of these values should be an offset; every following value is written to the next general purpose register, starting at that offset.

As the run control registers are accessed using memory requests on the PCIe bus, they can be performed relatively quick. Accesses to the other registers are implemented by performing DMA transfers, which require several memory requests and are thus slower.

The sysfs files are normally only accessible by the root user. To allow user space programs to access these files, several rules are added for the `udev` device manager[27]. These rules will change the group attribute of the files to the `rvex` group. Any program run by a user in the `rvex` group can then access the sysfs files.

4.3 pocl device-layer implementation

A new ρ-VEX device-layer has been added to the *Portable Computing Language (pocl)* framework. This layer has to perform the following functions: query devices, manage memory, transfer data, generate machine code, and manage execution. The implementation of the first three functions is straightforward and will be discussed first. Subsequently, a short overview is given of the techniques used by pocl to implement work-groups and the local address space. Finally, the generation of machine code and management of execution will be detailed.

Query devices To query the amount of available devices, the ρ-VEX device-layer scans the amount of `fpga` devices in the `rvex` class on sysfs using libudev. This allows adding multiple accelerators to a system, each being used by a different program.

Manage memory The management of the memory of the accelerator is done by the device-layer on the host. As a kernel instance can not dynamically allocate memory, the amount of memory that needs to be reserved is known before an instance is executed.

The ρ-VEX device-layer uses the `Bufalloc` memory allocator that is included in pocl. This allocator is designed for typical OpenCL workloads and uses a simple first fit algorithm to find free space[22]. As only one OpenCL program can use the device at the same time, the entire memory of the ML605 card is available to the allocator.

Transfer data Transferring data is done using the cdev interface of the `rvex` Linux driver. When the device-layer is initialized, the associated sysfs `rvex` device is stored. The associated character device is then used for read and write transfers.

Read and write requests are performed using the POSIX functions `read` and `write`. To set the location in memory where the data should be written, the `lseek` function is used. These functions are preferred over the standard C file functions defined in `stdio.h`, as the C functions have internal buffers.

In order to explain the generation of the machine code and management of the execution some additional information on the design of pocl is needed. In the following sections work-group functions, kernel signature transformation, argument lists, execution contexts, and the on-device setup will be discussed. Afterward, the machine code generation and execution management will be detailed.

4.3.1 Work-group functions

From the perspective of the programmer, all kernel instances in a work-group are executed concurrently. Instances in the same work-group can communicate with each other using local variables and wait on each other using barriers. A seemingly straightforward implementation would execute all work-groups in parallel. This becomes complicated and expensive when there are more instances in a work-group than the amount of compute units, as that would require scheduling and context-switching.

```
kernel void k(global int *glob_pointer, global image2d img,
    local int *local_pointer, long l) {
  local int a;
  ...
}

void k(global int* glob_pointer, global image2d img,
  local int* local_pointer, long l, local int* a, pocl_context* pctxt);
```

Figure 4.5: Transformation of signature by pocl framework.

In pocl, all kernel instances in a work-group are combined into a single instance, called a work-group function. pocl creates loops over all work-group items in “all parallel regions”, regions between barriers. For more information on the generation of work-group functions, see [22].

One of the main benefits of this technique is that a device can simply run work-group functions in parallel without having to implement shared memory and barriers, significantly simplifying the design of an accelerator. A large drawback is that a new work-group functions needs to be compiled for different work-group sizes. As the work-group size is only known when the kernel is enqueued, first-time execution of a kernel with a not yet used configuration might take multiple times longer than expected.

4.3.2 Kernel signature transformation

A kernel instance has two extra types of input in addition to its explicit arguments: variables declared in the local address space and the execution context. In pocl, these inputs are passed to the kernel function as extra arguments. This section describes this technique.

There are two types of variables that can be declared in the local address space. If a kernel argument is a pointer to a local variable, all kernel instances in the same work-group should get a pointer to the same variable. Variables declared on the stack and marked as local should also be shared among instances in the same work-group.

pocl adds an extra pass over the LLVM *Intermediate Representation (IR)* to extract local variables and add them as arguments to the kernel function. To illustrate the transformations that are performed, see the function definition and signature in Figure 4.5. A pointer to the local variable `a` has been appended to the arguments of the kernel. On execution, the framework will provide instances in the same work-group a pointer to the same variable.

The execution context describes the configuration of the index space and the index of the kernel instance. The kernel can access this information using the work-item functions: `get_global_id`, `get_global_size`, `get_group_id`, `get_local_id`, `get_local_size`, `get_num_groups`, and `get_work_dim`.

pocl adds a pointer to a `pocl_context` structure as the final argument to every kernel. This structure encodes the execution context and is used to implement the work-item functions.

4.3.3 Argument list

Before execution of the kernel, the arguments and local variables need to be copied to the device. To accomplish this, a region of memory is allocated to store these arguments. This region is called the argument list. The argument list consists of three parts: a list of pointers to the original arguments, a list of pointers to local variables declared in the kernel, and a buffer where the actual arguments are stored (the argument buffer).

Figure 4.6 contains an illustration of the argument list layout for the kernel defined in Figure 4.5. The first four entries are pointers to the location of the original four arguments. Of these, the first points to a buffer allocated by a previous `clCreateBuffer` call. This buffer is managed separately and should outlive the function. The image buffer itself is also managed separately and not located in the argument buffer. To correctly access the `image2d` object some metadata is needed though. This data is stored in a `dev_image_t` structure located on the argument buffer and contains a pointer to the image buffer.

Both the data pointed to by the `local int*` argument and the value of the `long` argument are stored in the argument buffer. The amount of memory to reserve for the third argument, the `local int*`, is determined by the application by calling the `clSetKernelArg` function. For Figure 4.6 the size of the argument is set to `sizeof(int) * 4`.

The `local int` variable `a` declared on the stack in Figure 4.5 will also be allocated in the argument buffer. As all kernel instances in the same work-group will be added to the same work-group function, each instance has access to the local data.

4.3.4 Execution Context

The structure of the execution context can be found in Listing 4.7. It consists of the following variables:

- `work_dim`: The amount of dimensions of the index space. A value between 1 and 3.
- `num_groups`: The number of work-groups of this execution, for each dimension.
- `group_id`: The identifier of the currently executing work-group function.
- `global_offset`: The offset at which global identifier start.

When the work-group functions are generated, the work-group local identifier is stored in a variable. The global identifier for a dimension $0 \leq d < 3$ is given by $\texttt{group\_id}[d] \cdot \texttt{num\_groups}[d] + \texttt{local\_id}[d] + \texttt{global\_offset}[d]$.

4.3.5 On-device setup

Before the kernel function can be called, the correct arguments need to be loaded. This is done by the work-group loader, of which the signature can be found in Listing 4.8. This loader reads the kernel function arguments from the argument list and calls the kernel.

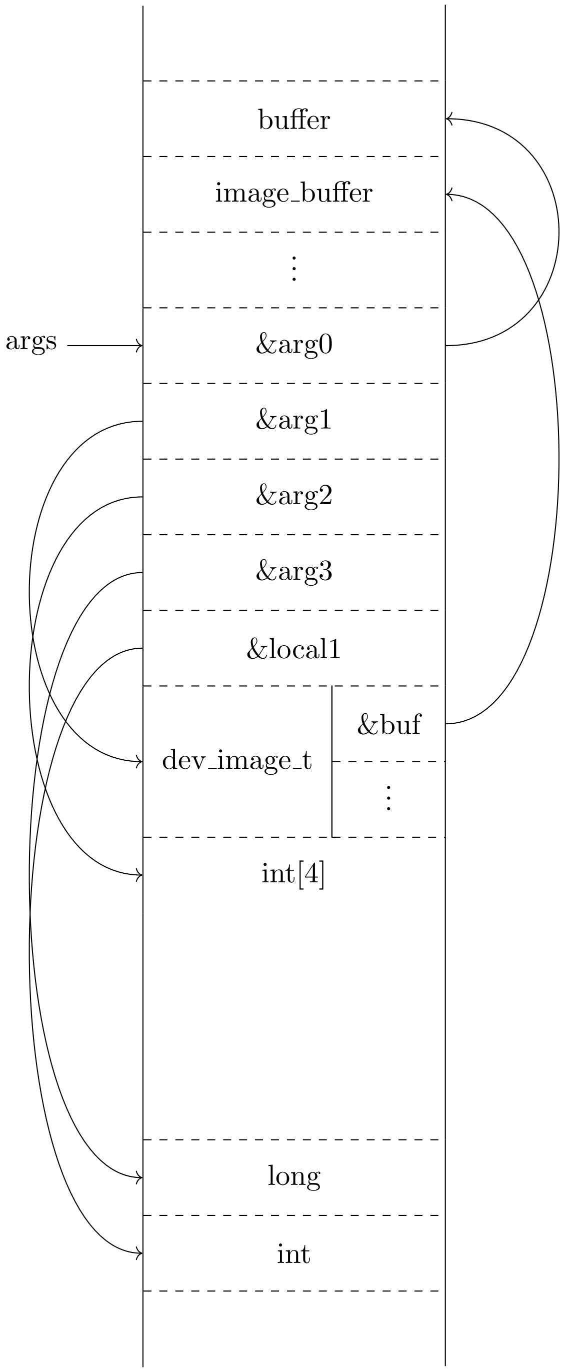

Figure 4.6: Memory layout of kernel arguments

As the amount and type of arguments differs between kernels, a special work-group loader is generated for each kernel.

The entry point of the program loaded onto the device is the `_start` procedure. This small procedure written in assembly initializes the start pointer, loads the correct

```
struct {
  uint32_t work_dim;
  uint32_t num_groups[3];
  uint32_t group_id[3];
  uint32_t global_offset[3];
};
```

Figure 4.7: Structure of the execution context

```
void k_workgroup_fast(void *arg_list, pocl_context *ctxt);
```

Figure 4.8: Signature of work-group loader function.

addresses of the argument list and execution context and calls the work-group loader. When the work-group loader returns the execution is halted, allowing the host to detect that the kernel is done.

In the current implementation of the device-layer, the addresses of the argument list and the execution context are stored in the first eight bytes of the kernel binary. This implementation requires multiple instances that are run concurrently to have a separate copy of the machine code in memory. When pointers to the argument list and execution context are passed in available scratchpad registers, a single version of the machine code can be used.

4.3.6 Generate machine code

Generating executable machine code for the ρ-VEX consists of two main phases: code generation and linking. The first step generates machine code from the LLVM IR, the second sets all global memory references to the correct value. This section explains the compile steps taken by pocl and the ρ-VEX device-layer.

In OpenCL, the kernel source code is specified to the framework using the `clCreateProgramWithSource` function. This source code is build upon a call to the `clBuildProgram` function. At this point, pocl will use Clang to compile the source code to LLVM IR. Extraction of local variables and transformation of the arguments is done when the `clCreateKernel` function is called. The machine code can only be generated when the `clEnqueueNDRangeKernel` is executed, as only at that moment the work-item size is known and the work-group function is generated.

To create machine code from the work-group function, first the LLVM VEX back end is executed. This will generate assembly code from the LLVM IR in which the work-group function is defined. Next, the assembly is turned into an object file by the assembler program. The `_start.s` file containing the entry point for the device is also assembled into an object file. Both object files are then linked together to create the machine code.

At the previous link step, the location in memory of the program is not yet known. This is required as the LLVM VEX back end currently does not support position-independent code generation and the ρ-VEX processor does not have a memory management unit. The location of the machine code in memory is known just before execution, when it is allocated. Before the code is transfered to the memory of the accelerator, the linker is run a final time to correctly set the memory location.

As described in Section 3.2, a development version of LLVM-TURBO has been evaluated for use as an LLVM back end. Because LLVM-TURBO is still in development, not all LLVM IR constructs are currently supported. As some of these unsupported constructs are used by pocl, the CoSy developed VEX back end could unfortunately not be used with the pocl ρ-VEX device-layer.

4.3.7 Manage execution

Before execution of kernel instances can start, the device needs to be prepared. Preparation consists of the following steps:

1. allocating memory for the argument list and the execution context
2. transferring the argument list to device memory
3. allocating memory for the kernel machine code
4. linking the kernel machine code to its final location
5. setting memory addresses of the argument list and execution context at start of kernel binary
6. transferring kernel binary to device memory

Special care has to be taken when the argument list and execution context are constructed. While the ρ-VEX processor is big-endian, the used host is little-endian. Any internal data needs to be converted to big-endian before it is transfered to the device.

The execution of the ρ-VEX processor is managed with the run control registers. After transferring the kernel binary to device memory, the `reset_vector` register is set to the address of the `_start` procedure. Before execution of each work-group function, the context is reset and the correct group identifier is set in the execution context. Next the context is allowed to run by setting the `run` signal to 1. To check if the execution is finished, the `done` signal is polled.

4.4 Conclusion

This chapter details the implementation of the ρ-VEX accelerator using OpenCL. It starts by describing the implementation of the PCIe interface on the accelerator platform, followed by the implementation of that interface on the host platform. Finally, the implementation of the ρ-VEX device-layer in pocl is explained.

For the ρ-VEX interface, the PCIe endpoint generated by the Xilinx CORE Generator System was used. Two components have been designed to interface between the DMA engine and the AMBA bus: one for transfers from the host to the accelerator and one for transfers from the accelerator to the host. Several execution control signals are connected to the register interface of the DMA engine, allowing quick control of the ρ-VEX processors on the accelerator.

For the host the `xdma` kernel driver is used to interface with the DMA engine on the FPGA. The original driver only supports 32-bit kernels and is designed for streaming applications. It has been modified to support 64-bit machines and support the addressed transfers needed by the accelerator.

A new kernel driver called `rvex` is developed to interface between the `xdma` driver and user space. To interface with the memory, a character device is created. File operations on this character device are translated to operations on the memory of the accelerator. Registers of the ρ-VEX processor on the accelerator are made available using the sysfs interface.

To add support for the ρ-VEX accelerator in pocl a new device-layer has been created. This device-layer defines device-specific functionality for the accelerator. The following functionality is implemented: querying the amount of devices, managing the memory of the device, transferring data to and from the device, generating machine code for the device and managing the execution of the device.

Experiments 5

This chapter describes the experiments performed to answer the research question. The goal is to create and validate a model of the run time of the accelerator on an example problem.

First the methodology and setup of the tests will be detailed. Subsequently, the results for the communication throughput and latency, cache misses, compile time, and kernel execution time will be given. Based on these results a model for the full execution time will be proposed and validated. Finally, the available compilers will be benchmarked to explore the improvement potential in that area.

5.1 Methodology

The performance of the accelerator depends on multiple factors. The identification of these factors is done by analyzing the different stages in the execution of an OpenCL program. For each stage, the performance factors are determined. Each of the factors differing between device-layers is measured separately. Based on these measurements, a total run-time of the program is predicted and validated.

To determine the relevant performance factors, seven execution stages of an OpenCL program are considered. An overview of these stages and the identified influencing factors is listed in Table 5.1. Each of these will now be shortly discussed.

The performance of the first three stages is mostly determined by the pocl framework. As the framework consists of many different parts, these parts can be listed individually. Stage 1 uses the generic probing and selecting API, and is the same for all accelerators. As stage 2 compiles the OpenCL files to LLVM *Intermediate Representation (IR)*, its

OpenCL execution stage	Performance factors
1) Query and select accelerators	Framework efficiency
2) Compile program OpenCL files	Clang execution speed
3) Compile specific kernel	LLVM VEX back end execution speed
4) Copy input data to device memory	Communication bus throughput, device and host memory throughput
5) Setup device for running kernel	Communication bus latency and throughput
6) Run kernel	Device clock frequency, device memory throughput and latency, compiled code quality
7) Copy result data to host memory	Communication bus throughput, device and host memory throughput

Table 5.1: Performance factors that influence execution stages of an OpenCL program

performance is determined by the execution speed of the Clang compiler. Stage 3 runs the LLVM VEX back end to generate machine code from the IR, so its performance is directly caused by the performance of the LLVM back end.

As most kernels require input data and create output data, copying data to and from the accelerator is an important part of the execution of an OpenCL program. This is done not only in stages 4 and 7, but also when copying the kernel machine code to the device in stage 5. The speed of copying is mostly determined by the throughput of the communication bus and the throughput of the host and device memory. As the data to be copied is normally large, latency is not a very large factor in the copy speed.

Apart from copying the machine code, several other operations have to be performed when setting up the device in stage 5. The program counter has to be pointing to the start of the kernel, and the correct parameters need to be passed to the kernel. This process requires several bus transfers between the accelerator and the host. Here, the latency of the communication bus between the host and the accelerator might have an influence.

Finally and obviously, the execution speed of the kernel on the accelerator is an important factor in the performance of the complete program. This execution speed is determined by architectural features such as the clock frequency of the accelerator, its memory throughput and latency, and cache misses. It is also impacted by the quality of the machine code generated by the compiler.

5.2 Setup

The performance of the accelerator is evaluated relative to the execution speed of running the kernel on the host *central processing unit (CPU)*. Where possible, tests are executed on both the host CPU using the pocl *basic* device-layer and on the accelerator using the pocl ρ-VEX device-layer.

The tests are run on a desktop machine with a dual core Intel Core 2 Duo E8500 Wolfdale processor with a maximum clock frequency of 3.16 GHz and 4 GB of DDR3 SDRAM. The desktop was running Ubuntu 14.10 64-bit with Linux kernel 3.16.

The accelerator is implemented on a Xilinx Virtex-6 FPGA on the Xilinx ML605 platform, which has 512 MB of DDR3 SDRAM. It contains one ρ-VEX processor. This processor has four contexts and can run up to four 2-issue threads, two 4-issue threads or one 8-issue thread. The processor is clocked at 37.5 MHz, as is the AMBA bus connecting the processor, the *Direct Memory Access (DMA)* engine and the memory. The accelerator contains a 1 kB data cache and an 8 kB instruction cache.

All experiments are run 5 times and the average of these 5 runs is taken. Where possible, experiments are run without the X.Org Server enabled to reduce the amount of context switches.

5.3 Communication

Communication throughput between the accelerator and the host over the *PCI Express (PCIe)* bus was measured by performing read and write requests of various sizes.

	Throughput (MB/s)	
Size	Reading	Writing
128 kB	6.58	10.59
256 kB	7.86	14.37
512 kB	8.87	17.24
1 MB	9.33	19.46
2 MB	9.61	20.75
4 MB	9.75	21.57
8 MB	9.83	22.06
16 MB	9.86	22.24
32 MB	9.84	22.41
64 MB	9.88	22.45

Table 5.2: Throughput over PCIe, using the cdev interface

The time it took to read or write was measured using the shell command `time`. The command for reading is then `time head -c $s /dev/fpga0 > /dev/null`, and for writing `time head -c $s /dev/zero > /dev/fpga0`, where `$s` is the size to read or write.

The measured throughput over the PCIe bus can be found in Table 5.2 and Figure 5.1. For large transfer sizes, the communication system achieves a bandwidth of 9.88 MB/s when reading and 22.45 MB/s when writing. The exact cause for the 2.3 times lower performance for reads relative to writes is not known. The likely problem is that due to the low frequency of the communication bus with the memory, data buffered from a previous read request is removed before the next request arives. This requires every read request to perform a complete memory access, leading to a large performance penalty. Writes are cached in the memory controller when received, so the rest of the system does not need to wait for the memory write to be completed.

To have an indication of the overhead of the `rvex` kernel module, the transfer times for different packet sizes are measured (see Table 5.3 and Figure 5.2). These times are measured from the moment the transfer is initiated by the host till the moment the card generates an interrupt indicating the completion of the transfer. As transfers larger than 4 kilobytes are split into smaller transfers by the driver, the measurements have been made for packets with a size up to 4096 bytes.

For reads of packet sizes up to 8 bytes, the transfer time is around 6.5 µs to 6.9 µs. As the throughput for large packets achieves more than 12 MB/s, the actual transfer of 8 bytes takes around 0.6 µs. This suggests a read latency of approximately 6 µs.

When looking at the transfer speed for writes, we see a linear relation between the size of a packet and its transfer speed. This relation holds for packets smaller than 1024 bytes, the speed for larger packets decreases. This suggests that there is a write buffer in the memory subsystem that can store approximately 1024 bytes. When four 1024 byte transfers are initiated at the same time, the transfers together complete in 135 ns. This is slightly slower than one 4096 byte transfer, as we expect due to the increased overhead associated with four transfers over one transfer.

The achieved throughput of 109 MB/s is close to 143 MB/s, the theoretical limit of the AMBA bus connecting the PCIe interface with the memory controller. This indicates

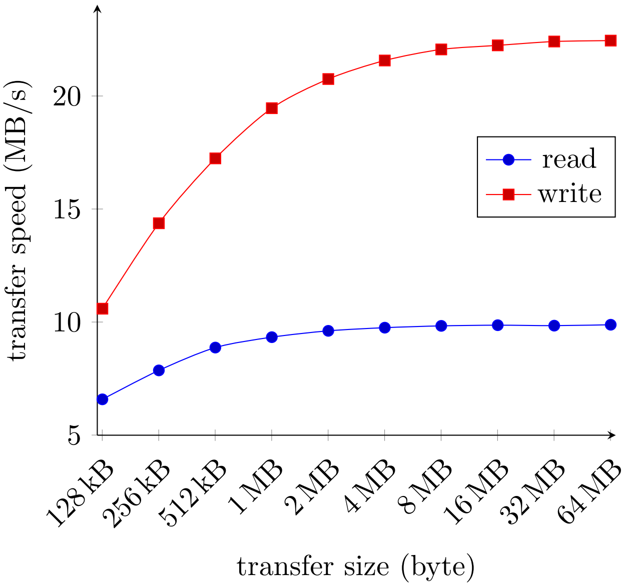

Figure 5.1: Throughput over PCIe for different transfer sizes, using the cdev interface

	Reading		Writing	
Size (bytes)	Time (ns)	Speed (MB/s)	Time (ns)	Speed (MB/s)
1	6663.0	0.15	9722.2	0.10
2	6928.2	0.29	6090.0	0.33
4	6565.0	0.61	6034.6	0.66
8	6481.4	1.23	8744.0	0.91
16	9093.4	1.76	6271.6	2.55
32	9903.6	3.23	8004.0	4.00
64	11523.8	5.55	7319.6	8.74
128	23117.8	5.54	9037.8	14.16
256	32253.2	7.94	9485.0	26.99
512	53541.0	9.56	9261.2	55.28
1024	93673.0	10.93	9373.2	109.25
2048	175779.4	11.65	33719.8	60.74
4096	337953.0	12.12	115533.6	35.45

Table 5.3: Transfer time and speed for different packet sizes over PCIe

that the PCIe and DMA interfaces can handle up to 109 MB/s per second, and that the limiting factor for the throughput on large packet sizes is the memory subsystem on the accelerator.

Given the measured PCIe transfer time of 12 MB/s for reading and 35 MB/s for writing 4 kB packets, we can determine the overhead of the Linux kernel driver interface. The overhead for reading is 18.5% and for writing is 40%.

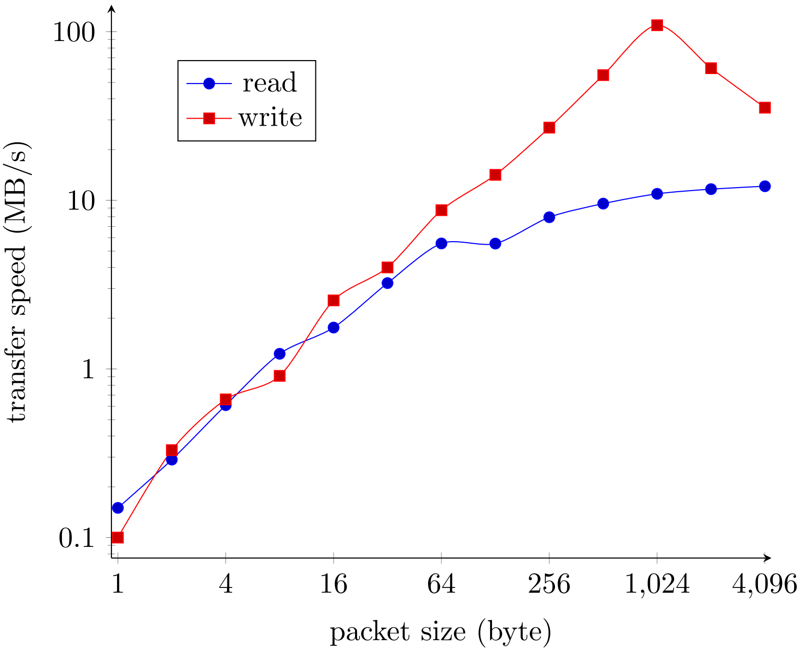

Figure 5.2: Transfer speed versus size when transferring a single packet over PCIe

5.4 Cache miss

Accessing the memory is an expensive operation on modern architectures[28]. To mitigate this problem, the accelerator has an instruction and data cache between the memory and the ρ-VEX processor. A memory access to cached data or instructions can be performed in a single cycle. Data or instructions not in the cache need to be retrieved from RAM, taking many cycles.

To measure the performance hit of a cache miss, an OpenCL kernel was written that would do one million 4-byte memory reads or writes. A cache miss on read was ensured by incrementing the address by 4 bytes after each read, as a cache-line in the cache configuration of the accelerator is only 4-bytes long. Because the data cache has only 256 cache lines, each read will be a cache miss. To reduce the amount of cycles spent on non-memory operations, 50 memory operations are performed per loop iteration.

A memory write will always be written to both the cache and the memory. As there is a write buffer in the ρ-VEX core, multiple writes have to be performed in consecutive cycles to measure the maximum latency of a write. Performing 50 consecutive writes per loop will reduce the effect of this write buffer, giving a good worst case estimate.

Using performance counters in the ρ-VEX, the total amount of cycles spent executing and the amount of stalled cycles have been measured. As can be seen in Table 5.4, a cache miss when reading stalls the core for 8 cycles. That the amount of executed cycles per read is slightly higher than 9 cycles is caused by loop overhead and sub-optimal scheduling. A write stalls the context 2.92 cycles. As writes are buffered, the actual stall is probably 3 cycles long, but the first write of the loop is buffered.

	Time (ns)	Cycles	Cycles stalled
Load	243.1	9.11	8.0
Store	106.8	4.01	2.92

Table 5.4: Read and write characteristics per operation

	Time (ms)		% stalled	
Processed images	rvex	basic	rvex	basic
1	14174	79.15	21.4	57.8
2	28346	157.95	21.4	57.6
3	42518	236.00	21.4	57.6
4	56690	314.60	21.4	57.6
5	70862	392.22	21.4	57.6
6	85035	468.87	21.4	57.5
7	99207	547.39	21.4	57.5
8	113379	625.75	21.4	57.5

Table 5.5: Execution time and percentage of stalled time of original kernel using ρ-VEX and host CPU

5.5 Kernel execution time

To measure the kernel execution time, an edge detection algorithm has been used. This algorithm performs a convolution on an image with a 3 by 3 pixel convolution matrix. The amount of images the edge-detection algorithm is run on is varied from 1 to 8 images, each with a dimension of 640 x 480 pixels. The code for the OpenCL kernel used can be found in Listing A.1 in Appendix A.

The execution time and percentage of stalled cycles during execution are listed in Table 5.5. As can be seen the accelerator is about 180 times slower than the host CPU. This is quite a bit more than expected from the clock speed difference alone, as the host CPU is clocked 84.4 times higher than the ρ-VEX processor on the accelerator.

While analyzing the generated assembly of the kernel, it was noted that it contained a lot of branches. This is caused by the use of the `read_imagei` and `write_imagei` methods. These methods can deal with different image types and check validity of the arguments. This functionality results in multiple branches, which significantly impact the *instruction-level parallelism (ILP)* and make techniques like loop unrolling less efficient.

To reduce the dependence on the compiler and kernel library quality, an optimized version of the kernel was tested (Listing A.2 in Appendix A). In this version, the knowledge of the structure of the image is used to directly access the pixel color values. The Clang front end is also asked to unroll the convolution loops using the *unroll* pragma. The front end can unroll these loops as the kernel sizes are known after function inlining. This unroll feature is not part of the OpenCL standard.

The measurement results for this kernel can be found in Table 5.6. The optimizations lead to a speedup of 9.3 on the accelerator and between the 4.4 and 5.0 on the host CPU. That difference in speedup can be explained by the increase in ILP that the compiler

	Time (ms)		% stalled	
Processed images	rvex	basic	rvex	basic
1	1520	17.84	26.7	43.8
2	3040	32.60	26.7	43.7
3	4560	47.73	26.7	43.9
4	6083	64.42	26.7	43.9
5	7604	79.39	26.7	44.0
6	9125	95.28	26.7	44.1
7	10646	110.44	26.7	44.1
8	12166	126.62	26.7	44.1

Table 5.6: Execution time and percentage of stalled time of optimized kernel using ρ-VEX and host CPU

can identify due to the reduced amount of branches in the new kernel and the unrolling of the convolution loop. As the super scaler architecture of the host CPU could already detect part of this ILP, its speedup is less than the speedup of the accelerator.

On the optimized kernel, the execution speed difference between the host CPU and the ρ-VEX has come closer to the expected speed based on clock frequency. The slowdown ranges from 85 times when processing one image to 96 times when processing eight images. The fact that the relative amount of stalled cycles has increased on the ρ-VEX can be explained by the reduction in overall executed instructions. As the amount of memory accesses doesn't decrease as much as the execution time, the relative amount of stalled cycles increases.

5.6 Compile time

As an OpenCL kernel is supplied as a source code string, it needs to be compiled to machine code before it can be executed. This step has to be done once, as the resulting machine code doesn't change unless the source code changes. Compiled binaries are cached by the framework, so the execution time is normally only effected during the first run of a program.

In pocl the compilation is separated in three steps: compilation to IR, generation of the work-group function, and instruction generation. While the IR should ideally be platform independent, certain language features differ per platform and result in different representations. Therefore, a separate IR file is generated for each platform. When the program requests a kernel to be created, pocl generates a work-group function for that kernel. This function handles the passing of arguments to the kernel and is the entry point of the kernel program on the device. Finally, the generated code IR is lowered to machine instructions and assembled.

Kernels compiled using the *basic* device-layer, which are run on the host processor, are compiled as position independent code. This allows them to run correctly irrespective of the memory location the instructions are loaded to. Because the LLVM VEX back end does not support generation of position independent code, an extra link step has to be

ρ-VEX				
compile	work-group function	instr. gen.	link	total
1746.4 ms	2129.3 ms	754.8 ms	28.8 ms	4659.4 ms

basic				
compile	work-group function	instr. gen.		total
2820.6 ms	2357.2 ms	302.4 ms		5480.2 ms

Table 5.7: Compilation and link times for OpenCL kernel

performed before loading the machine code. During this link step global memory offsets are calculated. Linking a kernel for the ρ-VEX has to be done before every upload, as only then the location of the machine code in memory is known.

Table 5.7 contains the compile and link times of the optimized edge-detection filter kernel. Compiling the kernel for the x64 architecture used by the host takes over a second longer than the compilation for the ρ-VEX. This suggests that the x64 back end performs more optimizations than the ρ-VEX back end.

5.7 Full program execution

Using the previous measurements of factors that influence the performance of an OpenCL program, a model is constructed to predict the execution time of a complete OpenCL program. This program performs an edge-detection algorithm (Figure A.2 in Appendix A) on a set of images, where each image has a dimension of 640 by 480 pixels. The program performs the following steps:

1. Load images from disk.
2. Select OpenCL device.
3. Transfer original images as image array to device.
4. Build kernel.
5. Setup device to run kernel.
6. Run kernel on images.
7. Transfer result images from device.
8. Store result images on disk.

Steps that can result in a different execution time between the ρ-VEX and the *basic* pocl device-layers are steps 3-7. As the images have 4 bytes per pixel and a dimension of 640 by 480 pixels, the size of one image is 1200 kB. The size of the generated machine code for the ρ-VEX kernel is 6.5 kB.

	Time (ms)	
Step	ρ-VEX	Host
Transfer original images to device	59.5	
Build kernel	4659.4	5480.2
Setup device to run kernel	29.6	
Run kernel on images	1520.8	15.8
Transfer result images from device	121.3	

Table 5.8: Expected time of steps in benchmark OpenCL program

This gives an expected transfer time of 59.5 ms per image to the accelerator and 121.3 ms per image from the accelerator. As was measured, generating the kernel binary takes 4659.4 ms and is only incurred the first time the program is executed. Setting up the kernel requires the linking and uploading of the kernel machine code. Linking the machine code takes 28.8 ms. The machine code is 6.5 kB, so the upload including overhead is expected to take approximately 0.8 ms. Finally, the kernel execution takes 1520.8 ms per image. An overview of these values can be found in Table 5.8.

These values give an expected execution time in milliseconds using the ρ-VEX device-layer for steps 3-7 of

$$29.6 + 4659.4c + 1701.6x$$

where x is the amount of images and c is 1 if the kernel is not cached and 0 otherwise.

For the *basic* device-layer, we assume that transfers are practically instant. Kernel compilation takes 5480.2 ms and execution takes on average 15.8 ms per image. The expected execution time is thus $5480.2c + 15.8x$. According to these models, the ρ-VEX is 107 times slower per image than the host.

Table 5.9 list the expected and measured total execution times for the accelerator. The faster execution of the accelerator compared to the expected execution suggests that some parameters are chosen too conservatively. The larger deviations in the uncached benchmarks might be caused by file system or other operating system caches. The maximum deviation between the expected and measured results stays within 4%.

The expected and measured execution times for the reference platform are listed in Table 5.10. It is clear that at least one factor that influences the execution time is not taken into account. This discrepancy has not been fully investigated, possible causes are the copying of the images or the loading of the shared library containing the kernel.

5.8 Compiler comparison

The execution time of the OpenCL kernels is determined for a large part by the quality of the compiler. The less efficient the assembly is the compiler generates, the more performant the accelerator has to be to reach the same execution time.

In this section an evaluation of the performance of the current compiler technology for the VEX is made. The five available compilers will be tested on both a broader benchmark suite (Powerstone) and the evaluation kernel discussed in Section 5.5. The first test is performed on a simulator, while the benchmark of the evaluation kernel is

	ρ-VEX					
	cached			uncached		
# images	expected	measured	difference	expected	measured	difference
1	1731.2	1709.0	-22.2	6390.6	6349.1	-41.5
2	3432.8	3408.1	-24.7	8092.2	8055.1	-37.1
3	5134.4	5112.6	-21.8	9793.8	9469.0	-324.8
4	6836.0	6808.1	-27.9	11495.4	11455.0	-40.4
5	8537.6	8509.9	-27.7	13197.0	13150.0	-47.0
6	10239.2	10205.7	-33.5	14898.6	14566.3	-332.3
7	11940.8	11900.3	-40.5	16600.2	16501.4	-98.8
8	13642.4	13595.7	-46.7	18301.8	18178.3	-123.5

Table 5.9: Predicted and measured performance on the accelerator

	basic					
	uncached			cached		
# images	expected	measured	difference	expected	measured	difference
1	15.8	29.5	13.7	5496.0	5515.5	19.5
2	31.6	49.4	17.8	5511.8	5541.4	29.6
3	47.4	65.4	18.0	5527.6	5558.5	30.9
4	63.2	82.7	19.5	5543.4	5574.7	31.3
5	79.0	98.9	19.9	5559.2	5586.7	27.5
6	94.8	117.0	22.2	5575.0	5605.0	30.0
7	110.6	133.0	22.4	5590.8	5624.6	33.8
8	126.4	151.0	24.6	5606.6	5636.2	29.6

Table 5.10: Predicted and measured performance on the CPU of the reference platform

performed on a synthesized design. To evaluate the ILP detection of the compilers, the evaluation kernel will also be tested in 2-issue mode.

5.8.1 Powerstone benchmark suite

To validate the performance of the VEX compiler created with CoSy (see Appendix B), a performance comparison of the available compilers has been made. A subset of the Powerstone benchmark suite[29] was used for this analysis. This suite consists of programs containing kernels of embedded applications from multiple domains. Only benchmarks using integer arithmetic have been tested, benchmarks using floating point numbers have not been included as the ρ-VEX does not have a floating point unit.

Some modifications have been made to the Powerstone benchmark programs. Several Powerstone benchmarks are platform dependent as they make assumptions on the size of integer types. These benchmarks have been rewritten to use platform independent types. The notification of the success of a benchmark has also been changed. The original benchmarks print a string to the standard output indicating if the result of the computation was correct. This has been changed to returning specific values on success

compiler	benchmark	optimization level	fault
GCC	adpcm	0	incorrect result
	bcnt	0	failed to finish
	blit	0	incorrect result
	engine	0	memory error
	g3fax	0	incorrect result
	jpeg	0	data error
	v42	0	failed to finish
LLVM	g3fax	1-3	failed to finish
	pocsag	1-3	failed to finish
Open64	crc	all	incorrect result
	engine	all	illegal instruction
	jpeg	1	failed to finish
	pocsag	all	incorrect result

Table 5.11: Compiler simulation failures

or failure.

The performance of the compiled benchmarks were measured by simulating execution on the xstsim simulator from STmicroelectronics. After execution the result value in register `$r0.3` is checked to ensure that the program was executed correctly. If the execution was correct, the amount of executed instruction bundles as reported by xstsim is recorded. The simulation is expected to complete within one minute, otherwise it is marked as "failed to finish".

All available VEX compilers are included in the comparison: HP, GCC, Open64, LLVM, CoSy and LLVM-TURBO. A binary is compiled for every supported optimization level of each compiler. The compilers were generating code for a 4-way core. To compare the compilers, the lowest amount of executed bundles was taken for each compiler from all optimization levels. This approach was used as more aggressive optimizations enable at higher optimization levels might reduce performance.

There were four reasons that measurements failed: incorrect results, failing to finish, memory errors and illegal instructions. The first two failures are often caused by scheduling bugs where delays between instructions with data dependencies are not correctly handled. These bugs can also lead to incorrect memory offsets, resulting in memory errors. The Open64 compiler uses an extended VEX ISA, producing several instructions that are not in the original *Instruction Set Architecture (ISA)*. While these instructions are not supported by the xstsim simulator, the ρ-VEX processor does support them.

An overview of simulation failures can be found in Table 5.11. The failures of GCC and LLVM have been found to be caused by compiler bugs. Not all failures of programs compiled by Open64 have been analyzed, but at least some are caused by unsupported instructions.

The best execution time of compiled benchmarks per compiler can be found in Table 5.12. An overview of the execution time of the compilers relative to the execution time of the HP compiler can be found in Figure 5.3. In this overview, configurations where a compiler failed on the higher optimization levels are not included. Still, slow-

	CoSy	GCC	HP	LLVM	LLVM-TURBO	Open64
adpcm	17826	20361	21368	23816	21127	22719
bcnt	1872	4574	1868	4616	5793	2548
blit	12178	15188	11614	17152	10440	6628
crc	9897	12904	12721	9152	11405	
engine	1524837	508942	626215	446250	1764676	
g3fax	698351	596062	472715	2182275	643269	459719
jpeg	1739315	1656048	1364382	1631745	2044126	1318880
pocsag	30283	25852	15721		27202	
ucbqsort	371677	272487	185897	269167	359636	274619
v42	2548684	2012647	1647725	2007788	2334386	2278508

Table 5.12: Fastest execution time of a benchmark per compiler

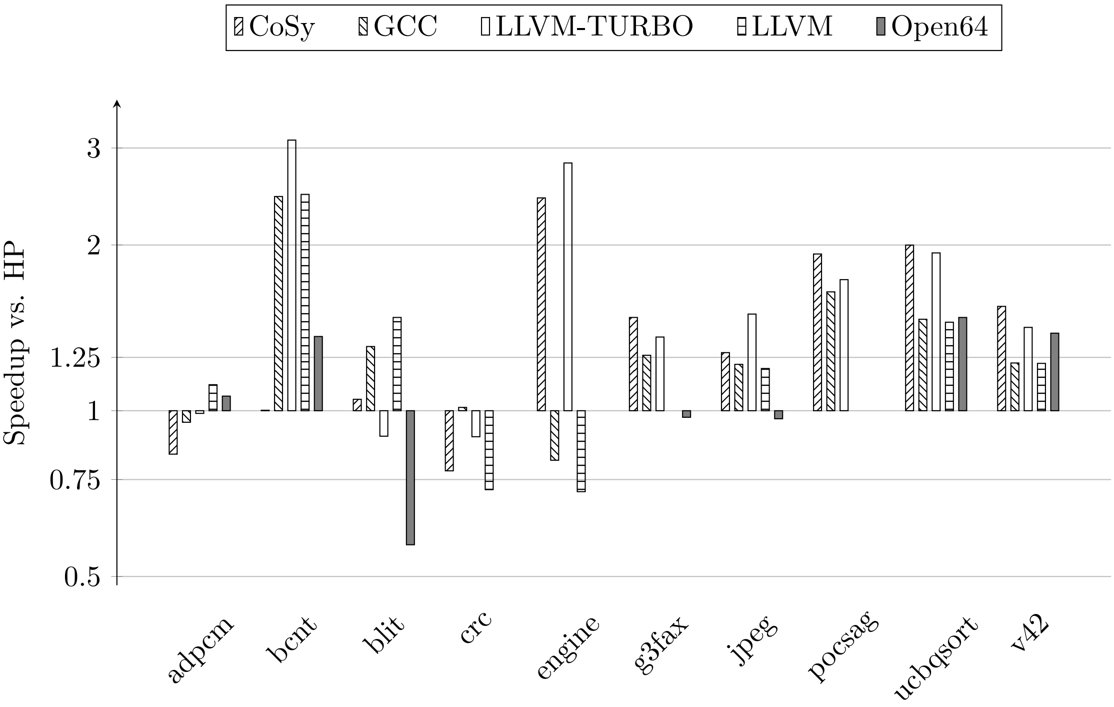

Figure 5.3: Speedup of benchmarks compiled with different compilers, relative to HP compiler. Results for the g3fax and pocsag benchmarks compiled with the LLVM compiler and the crc, engine, and pocsag benchmarks compiled with the Open64 compiler are not included.

downs of more than 1.5 occur with multiple compiler-benchmark configurations and only 12 of the 45 tested configurations are faster than the HP compiler. The bad performance of the CoSy back end based compilers on the engine benchmark is caused by inefficient division code. The causes for differences between compilers in the other benchmarks are currently not known and require more analysis.

compiler	optimization level	fault
CoSy	0, 1	failed to finish
LLVM-TURBO	1	failed to finish
LLVM	0-2	failed to finish
	4	incorrect result

Table 5.13: Compiler configurations not included in kernel execution time measurements

5.8.2 Performance on kernel

By comparing the performance of compilers on the Powerstone benchmark suite, it was found that there is large variation in the execution time of code generated by different compilers. In this section the impact of the compiler on the execution time of an OpenCL kernel is explored. This is done by compiling the kernel with all available compilers and analyzing the execution time.

The program that is used as a benchmark is based on the optimized kernel used in Section 5.5. This kernel is rewritten to be ANSI C compliant, allowing all compilers to process it. All functions needed by the kernel are included in the same source code file as the kernel, allowing the compilers to perform optimizations across functions. The main method and image data are located in separate source files so the compiler can't make assumptions on that data. As input data one of the 640 by 480 images is used that was also used when measuring the run-time of the OpenCL kernel. This image is added to the program as a constant variable, to make the binary self-contained.

The produced binaries are executed on a ρ-VEX soft core running on the ML605 platform. The ρ-VEX is configured in 8-way mode. To validate the correctness of the generated code a version of the benchmark program is used that checks the output of the kernel against a correct output. Programs taking longer than 5 minutes to complete or producing incorrect results are excluded from the results. The final measurements are performed with the output check disabled. Each measurement is performed five times and the average of these measurements is used.

Table 5.13 lists the configurations that where excluded from the measurements. Low optimization levels that fail to finish do not necessarily indicate a compiler bug. This is not the case for the incorrect result of the LLVM compiled binary, which is caused by a compiler bug.

The total amount of executed cycles and bundles can be found in Table 5.14. The most striking result is the really bad performance of CoSy and LLVM-TURBO, which both use the same VEX code generator. This bad performance is caused by the two modulo instructions in the inner loop of the convolution:

```
int imageX = (x - filter_size / 2 + filterX + imageWidth) % imageWidth;
int imageY = (y - filter_size / 2 + filterY + imageHeight) % imageHeight;
```

On lower optimization levels, the modulo operations are implemented as function calls, resulting in very low ILP. On higher optimization levels CoSy uses the fact that the right hand side of the modulo is a constant to generate code that should be significantly faster. This is not the case as that optimization uses operations on 64-bit values, which

	CoSy	GCC	HP	LLVM	LLVM-TURBO	Open64
bundle count ($\times 10^6$)	815.4	403.7	282.9	172.0	1065.3	67.0
cycle count ($\times 10^6$)	1152.0	428.1	343.5	198.6	2265.7	81.7

Table 5.14: Fastest execution time of all optimization levels on the optimized kernel

	CoSy	HP	LLVM-TURBO	Open64
slowdown	1.09	1.38	1.21	1.66

Table 5.15: Slowdown of a kernel compiled with an issue-width of 2 relative to a kernel with an issue-width of 8

are also implemented as function calls.

Performance differences between the other compilers are also large: the second fastest compiler (LLVM) generates code that is 2.4 times as slow as the fastest compiler (Open64). As the LLVM VEX back end is used by the pocl device layer, these results indicate that there is a lot of room for performance improvements by enhancing the LLVM compiler.

Impact of issue-width As a final test the impact of changing the issue-width of a kernel is explored. This is done by compiling the same kernel from the previous test for an issue-width of 2 instead of 8. As the GCC back end does not support a different issue-width than 8 and the code generated by the LLVM back end is incorrect, these compilers are not included in the results.

The slowdown of the kernel in a 2-issue configuration relative to an 8-issue configuration can be found in Table 5.15. The large slowdown of the kernel by the Open64 compiler suggests that that compiler is able to achieve a relatively high level of ILP. This would also explain the good performance of that compiler relative to the other compilers.

The slowdown is significantly smaller than the reduction in issue-width; a four-time reduction in issue-width leads to a slowdown of 9% to 66%. Although there is no data for the slowdown with the LLVM back end, the slowdown from the HP compiler can be used as an estimate. Running 4 kernels per ρ-VEX processor in a 4 times 2-issue configuration on the accelerator should thus result in an approximately 2.9 times higher throughput in executed instances.

5.9 Conclusion

This chapter describes measurements done to determine the performance of the current implementation of the accelerator, and the contribution of different factors to this performance. First the different performance factors are identified. Subsequently, the factors that differer between device-layers are measured independently. Using the measurements, a model of the expected run time of a benchmark kernel is formulated and validated. Finally, the impact of the compiler on the run time is analyzed by comparing the available compilers.

Measurement	Result
PCIe read throughput	9.88 MB/s
PCIe write throughput	22.45 MB/s
Cache miss time (read)	243.1 ns
Cache miss time (write)	106.8 ns
Kernel execution time	1520 ms
Compile time (total)	4659.4 ms
8-issue to 2-issue slowdown	1.09 to 1.66

Table 5.16: Overview of measurement results

The performance of the following components have been measured: PCIe throughput and latency with and without driver overhead, cache miss latencies, compilation time and kernel execution time (see Table 5.16). Where applicable similar measurements have been performed on the host CPU. Using these measurements a parameterized function is made to predict the execution time of the full program. This prediction is then validated by comparing it to the measured execution time. For the accelerator the prediction was found to be 95% accurate.

According to the proposed model of the execution time, there are three factors that influence the performance of the OpenCL implementation: the kernel execution time, the data transfer time and the compilation time. The kernel execution time was found to be 107 times slower per work-item on the ρ-VEX relative to the reference platform.

To analyze the influence of the compiler on the execution speed of its generated binaries, all available compilers for VEX were compared in two benchmarks. The first comparison uses the Powerstone benchmark suite. The large variation in the results indicates that the quality of the compiler has a large influence on the execution speed of the resulting program.

A second comparison is performed with the kernel used for the performance measurements of the accelerator. While the LLVM compiler was found to have the second best performance, the fastest compiler (Open64) was still 2.4 times faster than LLVM. Changing the configuration of the kernel from being compiled for an 8-issue processor to a 2-issue processor reduces the performance by 9% to 66%. Given that there are enough kernel instances to run, an accelerator running four 2-issue kernels in parallel could get a speedup of its overall execution time of 2.9.

6 Discussion

According to the measurements performed in the previous chapter, the developed accelerator is 107 times slower than the reference platform. This chapter discusses possible techniques to improve each of the factors identified in the model: the execution speed, the data transfer throughput and the compile time. Next, the effect of improving these factors on the total speedup is analyzed. Finally, a projection of the performance of the accelerator when implemented as an ASIC is made. These projected performance values are compared against both the reference platform and modern GPUs.

6.1 Execution speed

The performance of the ρ-VEX OpenCL accelerator is mostly limited by the execution speed of the kernel; one instance of the benchmark kernel took on average 15.8 ms on the host CPU and 1520.8 ms on the ρ-VEX. To attain performance parity on per kernel execution, the ρ-VEX processor needs to achieve a speed-up of 96 times. The time it takes to execute a program is given by the following equation [28]:

$$\textit{execution time} = \mathrm{IC} \times \left(\mathrm{CPI} + \frac{\textit{memory stall clock cycles}}{\textit{instruction}}\right) \times \textit{clock cycle time}$$

To lower the execution time of a program each of the variables in the equation can be reduced: the amount of instruction bundles needed to complete the program (instruction bundle count or IC), the amount of cycles executed per instruction (CPI), the amount of cycles stalled for memory, and the clock cycle time.

Reduce instruction bundle count One strategy to reduce the execution time of a program is to reduce the amount of instruction bundles needed to execute the program. This reduction can be achieved by improving the compiler, improving the kernel library, or by adding more specialized instructions.

By comparing the available compilers in Section 5.8 it was found that the Open64 compiler achieves a 2.4 times speedup over the LLVM compiler on the benchmark kernel. In other benchmarks the LLVM VEX showed large variations in its performance: it was the quickest in two benchmarks but between 11% and 147% slower on the others. While these are only a limited set of data points, it indicates that there is a room for improvement in the compiler currently used by the *Portable Computing Language (pocl)* device-layer.

Another way to reduce the IC is by optimizing the kernel library code. As described in Section 5.5, at least some of the library functions callable by an OpenCL kernel are implemented inefficiently. This was found to be the case for the functions used to read or write pixels to images. One way to optimize the generated code for these functions is by creating optimization passes in the compiler to remove checks known to be correct at compile-time.

The final way to reduce the IC is by adding extra instructions to the VEX *Instruction Set Architecture (ISA)*. If these instructions can perform operations with a single instruction that would otherwise require multiple instructions, the total amount of instruction bundles needed is reduced. Adding such instructions will require compiler or library changes so kernels are generated that will use the new instructions.

Reduce cycles per instruction There are two factors that influence the amount of cycles per instruction: the amount of scheduled instructions per bundle and the delay of an instruction before its result can be used. Increasing the amount of instruction per bundle would decrease the CPI, as executing a bundle takes one clock cycle[1]. This can be done by optimizing the compiler to create higher *instruction-level parallelism (ILP)*.

In Section 5.8.2 the slowdown found between a kernel compiled in a 2-issue configuration and an 8-issue configuration indicates that the found ILP is quite low. While this might be partly a property of the tested kernel, the variation in slowdown between the different compilers indicate that there is ILP not found by all compilers.

Alternatively, the CPI can be reduced by modifying the core to reduce the delays required between instructions. No analysis has been made of the reduction that can be achieved with this technique.

Reduce memory stall cycles Memory stalls occur when there is a read from a non-cached address or when a write is performed before the previous write is finished. The amount of cycles that the memory is stalled can be reduced by either increasing the cache and write buffer size or by reducing the memory latency.

In Section 5.4 it was found that a read cache miss takes 243.1 ns and a write when the write buffer is full takes 106.8 ns. A DDR3-1333 memory with CL and RCD timings 9 has a memory access time of at least $t_{RCD} + t_{CL} = 27$ ns [30]. The found latencies are several times higher than than these numbers, suggesting that more performance should be possible with the same memory. No measurements have been performed to determine what the cause is of the difference between the measured and the expected results.

Reduce clock cycle time As one bundle can be executed each cycle, reducing the time of a clock cycle will increase the amount of bundles and thus instructions executed per second. Instead of clock cycle time often the term for the inverse is used: the clock frequency.

On the optimized kernel, the ρ-VEX processor is on average 96 times slower than the host CPU (see Section 5.5). This is 88% slower per clock cycle when correcting for the difference in clock frequency. Simply increasing the clock frequency of the ρ-VEX processor by 96 times to 3.6 GHz will *not* result in performance parity though.

When the clock frequency is increased, and thus the clock cycle time is reduced, the time it takes to perform a memory access does not change. Increasing the clock frequency will thus lead to an increase of the amount of stall clock cycles per instruction. Without

[1] Except on a memory stall or a pipeline stall due to a branch.

a decrease in the amount of stalled memory cycles, it is not possible to get the execution time of the ρ-VEX processor on the example kernel to match the time of the host *central processing unit (CPU)*.

The maximum achievable clock frequency is determined by the longest signal path in the hardware design. If a signal change can not reach its destination register before the end of the cycle, an incorrect value is stored in the register and will propagate through the design. Shortening the longest signal path can be done by optimizing the placement and routing, potentially by changing the schematic design. Alternatively, a different lithography process with a smaller transistor size can be used for production. As the transistors can be placed closer together, the length of the signal paths decreases and the clock frequency can be increased.

6.2 Concurrently executing kernel instances

An alternative to decreasing the execution time of a kernel instance is to increase the amount of instances running simultaneously. As OpenCL is designed to be used for applications with high data parallelism, increasing the amount of instances being able to execute in parallel should lead to a near-linear speedup.

The simplest way to increase the amount of concurrently executing kernels is by compiling the kernel in a 2-issue configuration and running four 2-issue instances on a ρ-VEX core. From the comparison between 2-issue and 8-issue kernels made in Section 5.8.2 it follows that the execution time of four 2-issue instances run in parallel is lower than the four 8-issue instances run sequentially.

If enough transistors are available on a die, an accelerator can be made that contains multiple ρ-VEX processors. Each of these processors can run up to 4 kernel instances independently of each other. In such an architecture the size and location of caches becomes even more important, as otherwise the bus to the memory becomes a bottleneck.

6.3 Data transfer

When the execution time of the kernel instances is reduced, the data transfer time over the *PCI Express (PCIe)* bus will become a larger part of the total execution time. Reducing this transfer time will be required to achieve performance parity with the host CPU. On the current design, transferring one image to and from the ρ-VEX takes 181 ms, giving a throughput of 12.9 MB/s (see Section 5.3). This transfer takes in the order of a millisecond on the host.

The maximum memory throughput on the ML605 is currently limited by the memory subsystem. The AMBA bus connecting the PCIe bus with the memory system runs at 37.5 MHz and can transfer 4 bytes per cycle, giving a maximum throughput of 143 MB/s. Transfer times approaching this maximum throughput were measured, indicating that the throughput is limited by the memory subsystem. As the DDR3 SDRAM on the ML605 should be able to achieve bandwidths in the order of gigabytes per second on sequential accesses, the observed performance is probably caused by an incorrect configuration. For a throughput larger than 143 MB/s, the frequency of the AMBA bus needs to be increased.

The accelerator uses a version one PCIe link in with 4 lanes. The maximum throughput of such a link is 1 GB/s for transmit and 1 GB/s for receiving. If required, higher throughput is available by using more lanes or newer PCIe versions. The latest generation PCIe, version 3, enables a maximum throughput of 31.51 GB/s in each direction when configured with 16 lanes[16].

6.4 Compile time

The last factor that determines the total execution time of the program is the time it takes to compile the kernels. On the benchmark kernel, the compilation steps take 4.7 seconds (see Section 5.6).

While the compile time can become large when many kernels are used, the resulting binaries are cached. The overhead is therefore only incurred on installation or first run of the program on a system. In a normal OpenCL program, a single kernel is run many times. When the compile time of a kernel is amortized over the number of times the kernel is run, the impact of the compile time will be small.

For some applications, the impact of compiling the kernels the first time the program is executed might be to big. In that case the kernels can be compiled ahead of time, for instance during start-up of the program or during installation. If all targeted devices are known during development, compiled versions of the kernels can also be shipped with the program.

6.5 Total speedup analysis

As was noted by Amdahl[31], if the execution time of a program consists of several different factors, increasing the performance of only one of the factors will result in a sub-linear increase of the total performance. To analyze this effect on the model for the full program execution time proposed in Section 5.7, an exploration of the search space has been made. This exploration analyzes the attainable *total speedup* (S_t) by varying the *frequency speedup* (S_f), *data transfer speedup* (S_d) and *memory stall cycle reduction* (S_m).

The results of this exploration can be found in Figure 6.1. In this exploration the upper limit of S_f is chosen at 100, giving 3.75 GHz. Only improving the frequency will result in nearly no improvement for $S_f > 10$. The maximum total speedup achievable by only changing S_f is 2.90.

Data transfer speedup values of 3, 10, and 100 have been tried. Only the configuration with $S_d = 100$ is shown in the figure, as all other values are to close to the $S_m = S_d = 1$ configuration. By changing both S_f and S_d, a maximal theoretical total speedup is achievable of 4.17.

Finally, two reduction values have been considered for the time spent when stalled waiting for memory: $S_m = 4$ and $S_m = 9$. These speedups are based on the theoretical maximum performance of the DDR3 SDRAM memory, and shouldn't require adding more caches. From Figure 6.1 it can be seen that reducing the amount of memory stall

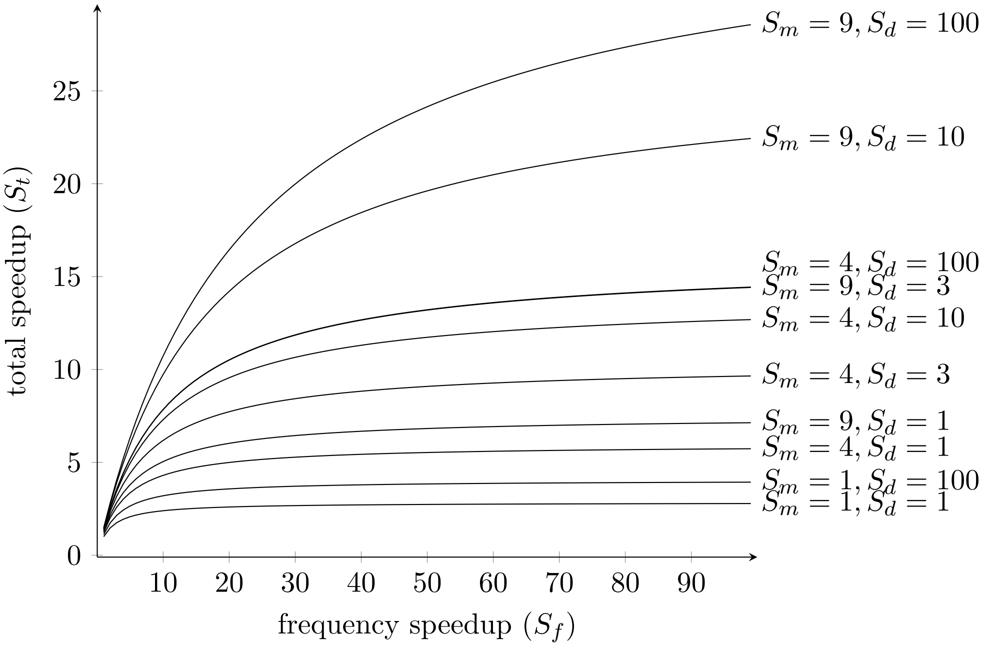

Figure 6.1: Speedup of total execution time on benchmark kernel, for varying data transfer rate increases and memory stall cycle decreases. Note that the lines for $S_m = 4, S_d = 100$ and $S_m = 9, S_d = 3$ are superimposed.

cycles has a large positive impact on the total performance. This is illustrated by the $S_m = 4, S_d = 100$ and $S_m = 9, S_d = 3$ configurations. There, a $2^1/_4$ times reduction in the amount of stall cycles leads to a similar total performance behavior as the $33^1/_3$ times speedup in transfer time.

6.6 Performance projection and comparison

In the previous sections techniques were discussed to increase the performance of the overall program. This was done by influencing five different factors: instruction bundle count, cycles per instruction, memory stall cycles, clock frequency and the amount of cores. In this section a projection is made of the attainable speedup by implementing the accelerator on an *application-specific integrated circuit (ASIC)* using a modern fabrication process. This projection is then compared to modern CPUs and *graphics processing units (GPUs)*.

In the previous section an exploration of the total performance behavior was made by improving the clock frequency, the data transfer throughput, and the amount of memory stall cycles. For the coming discussion, a reduction in the amount of stall cycles of between $4 \leq S_m \leq 9$ is used.

To create a projection of the amount of cores that fit on a die, an estimate of the area of the ρ-VEX processor on a die is needed. To get an approximation of this area, the area of the ST200-STB1 processor is used. This 4-issue *Very Long Instruction Word (VLIW)*

manufacturer	device	node (nm)	die size (mm^2)
AMD	Radeon HD5870 [36]	40	334
Intel	i7 920 [37]	45	263
NVIDIA	Tesla C1060 [38][39]	55	470

Table 6.1: Production processes of devices tested by Komatsu et al.

processor is based on the VEX ISA, has two 32 kB data and instruction caches and has a clock frequency of 300 MHz. According to [32], this processor has a core of 5 mm^2 and is produced on 250 nm. While not explicitly mentioned, it is implicated that the two 32 kB caches together are also 5 mm^2.

It is assumed that the area of a component on a die shrinks quadratically with the size of a transistor.

Comparison with reference platform The CPU used in the reference platform is the Intel Core 2 Duo E8500 Wolfdale. This processor is produced on 45 nm and has a die area of 107 mm^2[33]. Using the RAMspeed benchmark[34], a sustained sequential memory throughput of 2.4 GB has been measured. This gives a data transfer speedup of $S_d = 200$.

On 45 nm the ρ-VEX processor with two 32 kB caches would use an area of approximately 0.3 mm^2. If the same die area is available as is used by an E8500, 200 ρ-VEX cores could be placed. This would leave 47 mm^2 for extra caches, a PCIe interface, a memory controller, and an interconnect.

Given the step in production technology, it is assumed that the ρ-VEX processor can run at 300 MHz. This would mean a speedup of 8 over the clock frequency of the prototype, giving an expected total speedup per core of 7.0 to 9.2.

Executing kernel instances on 200 cores will result in memory contention and other overhead. For this discussion we will assume this overhead to be somewhere from 0% to 100%. The total speedup of a 200 core accelerator with a similar production process as the Intel E8500 would than be $5.1 \cdot 10^2 < S_t < 8.1 \cdot 10^2$ for $S_m = 4$ and $6.2 \cdot 10^2 < S_t < 9.3 \cdot 10^2$ for $S_m = 9$. Relative to the reference platform, that would be a speedup of $5.4 < S_t < 8.4$ for $S_m = 4$ and $6.4 < S_t < 9.7$ for $S_m = 9$.

An additional speedup can be achieved by moving to a four times 2-issue configuration for each ρ-VEX processor. With the found 2.9 times speedup, relative to the reference the total speedup would be between 10 and 15. The speedup for different overhead values can be found in Figure 6.2.

Comparison with GPUs The currently most used OpenCL accelerators are GPUs. To get an indication of the potential of an accelerator using the ρ-VEX processor, the projected performance is compared with two modern GPUs.

Komatsu et al.[35] compared an Intel Core i7 920 CPU with two GPUs: the NVIDIA Tesla C1060 and the AMD Radeon HD5870. In their tests they found the sustained performance of the GPUs to be between 20 - 183 times higher than the CPU. The production features of these devices can be found in Table 6.1.

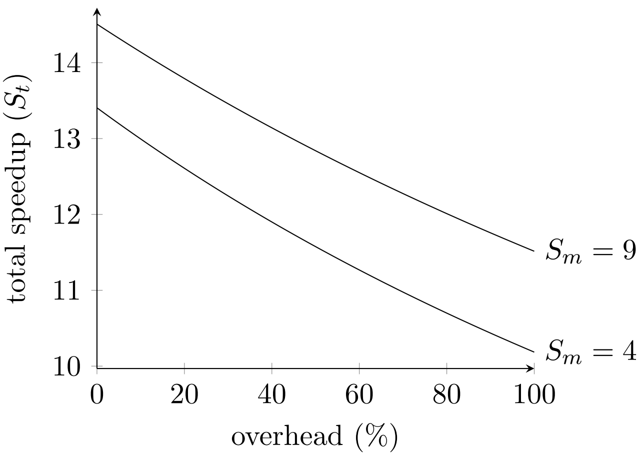

Figure 6.2: Projection of total execution time speedup when produced on same technology as the Core 2 Duo E8500 processor.

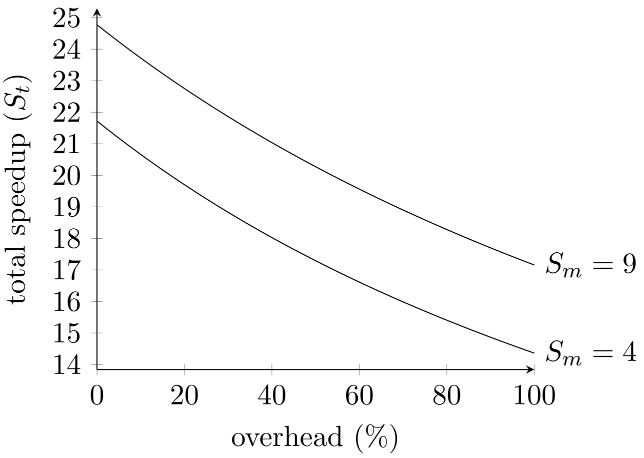

Figure 6.3: Projection of total execution time speedup when produced on same technology as the Core i7 920 processor.

According to [40], the Intel Core i7 920 is 2.2 times faster than the Intel Core 2 Duo E8500. Both devices are produced on the same node and the die size increased 2.5 times from the E8500 to the 920. As this die size increase is close to the speed increase, it is assumed that the execution speedup values found in the previous comparison will also hold against the Core i7 when produced on that process.

In their paper, Komatsu et al.[35] indicate that the memory bandwidth of the Core i7 is 25.6 GB/s. As it is not clear if this is a theoretical or measured bandwidth, half of that value is used for this projection, giving a data transfer speedup of $S_d = 1000$.

Assuming that the overhead will stay below 10%, the accelerator would be between 1.2 times faster and 8.8 times slower than the measured GPUs. The speedup results relative to the Core i7 can be found in Figure 6.3.

GPUs have multiple processing units that are controlled by a single *Program Counter (PC)*. Branches are handled by disabling processing units on which the instructions should not be executed. Conversely, the ρ-VEX processor has a separate PC for each context. It is therefore expected that many branch-heavy kernels execute faster on an accelerator based on the ρ-VEX processor then on a GPU.

For the projections no major architectural changes have been assumed. With such changes, further improvements might be possible. There are several improvements that can be made to the memory architecture, such as increasing the amount of cache, increasing the size of cache lines, adding multiple levels of cache, or creating use managed local memories. These changes should reduce the amount of cache misses, resulting in a reduced memory stall time. Increases in clock frequency can be achieved by creating a deeper pipeline.

Above projections are made the results of one benchmark and several assumptions and extrapolations. As motivated, the type of kernel instance can have a large impact on the performance of an accelerator. Further research is needed to analyze how different types of kernels perform on a ρ-VEX accelerator and to validate the projections.

6.7 Conclusion

This chapter discusses the measurement results of Chapter 5. First, the factors that influence the performance according to the found model are analyzed. Subsequently, the effect of improving these factors on the total speedup is analyzed. Finally, a projection of the performance of the accelerator when implemented as an ASIC is made. These projected performance values are compared against both the reference platform and modern GPUs.

There are four factors that influence the execution time of a single kernel: the amount of instruction bundles needed to complete the program, the amount of cycles executed per instruction, the amount of memory stall cycles, and the clock cycle time. An overview of the different techniques that can be used to improve the execution time can be found in Table 6.2.

Reducing the amount of instruction bundles needed to complete the program can be achieved by improving the compiler, improving the kernel library, or by adding specialized instructions. The amount of cycles executed per instruction can be reduced by optimizing the ILP detection of compilers or by reducing the required delay between instructions. Improving the amount of cycles spent on memory stalls can be done by changing the cache architecture or by reducing the memory access latency. To reduce the clock cycle time the longest signal path needs to be shortened. This can be achieved by optimizing the placement and routing, changing the processor design or moving to a lithography process with smaller transistor sizes.

As an alternative to decreasing the execution time of kernel instances, multiple instances can be run concurrently. This can be achieved by configuring the ρ-VEX processor in a four times 2-issue configuration or by placing multiple cores on the accelerator.

Data transfers throughput of the current design is 12.1 MB/s. Measurements suggest that this throughput is limited by the performance of the memory controller. For a throughput larger than 150 MB/s, the frequency of the memory bus needs to be increased.

Factor	Improvement	Technique
execution speed	reduce IC	- improve compiler - optimize kernel library code - extend ISA
	reduce CPI	- improve compiler ILP detection - reduce instruction delays
	reduce memory stall cycles	- increase cache - reduce memory access latencies
	reduce clock cycle time	- optimize processor design - reduce transistor size
	increase parallel execution	- 4 × 2-issue instead of 1 × 8-issue - multiple processors on an accelerator
data transfer	increase data throughput	- change memory subsystem configuration - increase AMBA bus frequency - change PCIe configuration
compile time	reduce compile time impact	- compile ahead of time

Table 6.2: Techniques to improve factors that influence the execution time

While compiling the kernel takes several seconds, it needs to be done only once per machine. As a kernel is normally run many times over the installed time of a program, the amortized cost of compiling the kernel is very low.

An exploration has been made of the effect that improving the different factors of the model has on the total speedup. In the exploration the effect of improving the frequency was analyzed for data transfer speedups of 1, 3, 10, and 100, and memory stall cycle reductions of 1, 4, and 9. It was found that if only the frequency is improved a maximal total speedup of 2.90 can be achieved and if both the frequency and the data transfer is improved the maximal total speedup is 4.17.

To compare a production version of the accelerator with current consumer products, two projections have been made. For these projections, a reduction of 4 to 9 times for the amount of memory stall cycles has been used. As approximation of the area used by the ρ-VEX processor, the size of the ST200-STB1 is used. By correcting for the change in production process, 200 ρ-VEX processors can be placed on an accelerator. With a data transfer speedup of 200, the projected total speedup over the reference E8500 CPU is 10 to 15. In comparison to GPUs, the projected speedup is between 1.2 times faster and 8.8 times slower.

Due to architectural differences between an accelerator containing many ρ-VEX processors and a GPU, kernels that contain many branches are expected to achieve better performance on the ρ-VEX based accelerator. Further research is needed to determine applications with these type of kernels. Another area for future research is the architecture of the accelerator; it is expected that improvements to the cache and pipeline architecture can further improve the performance of the accelerator.

Conclusion 7

7.1 Summary

Chapter 2 presents the different technologies necessary to understand the rest of the thesis. It starts with an overview of the ρ-VEX processor, which is a processor with a run-time reconfigurable issue-width and core count. ρ-VEX processors are currently instantiated as a soft-core on a *field-programmable gate array (FPGA)*. The GRLIB IP library is used to provide access to DDR3 *Random Access Memory (RAM)* on the Xilinx ML605 platform. A rich tool chain is available for the ρ-VEX processor. It consists of multiple compilers, a port of the GNU binutils project, and debugging tools.

As the created accelerator uses the *Open Computing Language (OpenCL)* framework, a short introduction to the OpenCL framework is given. This technology defines a framework for programs to execute kernels on multiple compute units. These kernels are shipped as source code and compiled by the framework implementation for the correct architecture. Several other compute related technologies are also mentioned.

Finally, a short overview of the functioning of the *PCI Express (PCIe)* bus is given. This bus provides a high-speed interconnect between peripherals and the *central processing unit (CPU)*. It allows mapping parts of the system memory range to a device for direct access by the CPU. Additionally, devices can perform *Direct Memory Access (DMA)* by accessing system memory without involving the CPU.

Chapter 3 presents the concept for the OpenCL accelerator implementation. It starts by giving an overview of the entire system, followed by descriptions of the components: the OpenCL runtime, the accelerator platform, and the communication interface.

As OpenCL runtime the open-source implementation *Portable Computing Language (pocl)* is chosen, as it is specifically designed to be used with architecturally different accelerators. To add support for the ρ-VEX based accelerator, a new device-layer is added to pocl. To compile the LLVM *Intermediate Representation (IR)* to VEX assembly, the LLVM back end developed by Daverveldt will be used. Additionally, an evaluation version of the LLVM-TURBO technology will be evaluated for use with pocl.

The Xilinx ML605 platform will be used for the accelerator. The ρ-VEX processor will use the GRLIB configuration, as it provides a memory controller. To implement DMA transactions an IP core by Northwest Logic is used. This DMA interface gives the *Program Counter (PC)* access to the bus connecting the core to the memory and other peripherals. A second interface using PCIe registers is available to control and monitor the execution of contexts on the core.

On the host the Xilinx DMA driver is used to handle DMA transactions. This driver is used by a new driver (`rvex`) that provides user space applications an interface to interact with the core. Access to the accelerator memory is given using a character device, while registers and execution control signals are available over the sysfs interface.

Chapter 4 details the implementation of the ρ-VEX accelerator using OpenCL. It starts by describing the implementation of the PCIe interface on the accelerator platform, followed by the implementation of that interface on the host platform. Finally, the implementation of the ρ-VEX device-layer in pocl is explained.

For the ρ-VEX interface, the PCIe endpoint generated by the Xilinx CORE Generator System was used. Two components have been designed to interface between the DMA engine and the *Advanced Microcontroller Bus Architecture (AMBA)* bus: one for transfers from the host to the accelerator and one for transfers from the accelerator to the host. Several execution control signals are connected to the register interface of the DMA engine, allowing quick control of the ρ-VEX processors on the accelerator.

For the host the `xdma` kernel driver is used to interface with the DMA engine on the FPGA. The original driver only supports 32-bit kernels and is designed for streaming applications. It has been modified to support 64-bit machines and support the addressed transfers needed by the accelerator.

A new kernel driver called `rvex` is developed to interface between the `xdma` driver and user space. To interface with the memory, a character device is created. File operations on this character device are translated to operations on the memory of the accelerator. Registers of the ρ-VEX processor on the accelerator are made available using the sysfs interface.

To add support for the ρ-VEX accelerator in pocl a new device-layer has been created. This device-layer defines device-specific functionality for the accelerator. The following functionality is implemented: querying the amount of devices, managing the memory of the device, transferring data to and from the device, generating machine code for the device and managing the execution of the device.

Chapter 5 describes measurements done to determine the performance of the current implementation of the accelerator, and the contribution of different factors to this performance. First the different performance factors are identified. Subsequently, the factors that differer between device-layers are measured independently. Using the measurements, a model of the expected run time of a benchmark kernel is formulated and validated. Finally, the impact of the compiler on the run time is analyzed by comparing the available compilers.

The performance of the following components have been measured: PCIe throughput and latency with and without driver overhead, cache miss latencies, compilation time and kernel execution time. Where applicable similar measurements have been performed on the host CPU. Using these measurements a parameterized function is made to predict the execution time of the full program. This prediction is then validated by comparing it to the measured execution time. For the accelerator the prediction was found to be 95% accurate.

According to the proposed model of the execution time, there are three factors that influence the performance of the OpenCL implementation: the kernel execution time, the data transfer time and the compilation time. The kernel execution time was found to be 107 times slower per work-item on the ρ-VEX relative to the reference platform.

To analyze the influence of the compiler on the execution speed of its generated binaries, all available compilers for VEX were compared in two benchmarks. The first comparison uses the Powerstone benchmark suite. The large variation in the results indicates that the quality of the compiler has a large influence on the execution speed of the resulting program.

A second comparison is performed with the kernel used for the performance measurements of the accelerator. While the LLVM compiler was found to have the second best performance, the fastest compiler (Open64) was still 2.4 times faster than LLVM. Changing the configuration of the kernel from being compiled for an 8-issue processor to a 2-issue processor reduces the performance by 9% to 66%. Given that there are enough kernel instances to run, an accelerator running four 2-issue kernels in parallel could get a speedup of its overall execution time of 2.9.

Chapter 6 discusses the measurement results of the previous chapter. First, the factors that influence the performance according to the found model are analyzed. Subsequently, the effect of improving these factors on the total speedup is analyzed. Finally, a projection of the performance of the accelerator when implemented as an ASIC is made. These projected performance values are compared against both the reference platform and modern GPUs.

There are four factors that influence the execution time of a single kernel: the amount of instruction bundles needed to complete the program, the amount of cycles executed per instruction, the amount of memory stall cycles, and the clock cycle time.

Reducing the amount of instruction bundles needed to complete the program can be achieved by improving the compiler, improving the kernel library, or by adding specialized instructions. The amount of cycles executed per instruction can be reduced by optimizing the *instruction-level parallelism (ILP)* detection of compilers or by reducing the required delay between instructions. Improving the amount of cycles spent on memory stalls can be done by changing the cache architecture or by reducing the memory access latency. To reduce the clock cycle time the longest signal path needs to be shortened. This can be achieved by optimizing the placement and routing, changing the processor design or moving to a lithography process with smaller transistor sizes.

As an alternative to decreasing the execution time of kernel instances, multiple instances can be run concurrently. This can be achieved by configuring the ρ-VEX processor in a four times 2-issue configuration or by placing multiple cores on the accelerator.

Data transfers throughput of the current design is 12.1 MB/s. Measurements suggest that this throughput is limited by the performance of the memory controller. For a throughput larger than 150 MB/s, the frequency of the memory bus needs to be increased.

While compiling the kernel takes several seconds, it needs to be done only once per machine. As a kernel is normally run many times over the installed time of a program, the amortized cost of compiling the kernel is very low.

An exploration has been made of the effect that improving the different factors of the model has on the total speedup. In the exploration the effect of improving the frequency was analyzed for data transfer speedups of 1, 3, 10, and 100, and memory stall cycle reductions of 1, 4, and 9. It was found that if only the frequency is improved a maximal total speedup of 2.90 can be achieved and if both the frequency and the data transfer is improved the maximal total speedup is 4.17.

To compare a production version of the accelerator with current consumer products, two projections have been made. For these projections, a reduction of 4 to 9 times for the amount of memory stall cycles has been used. As approximation of the area used by the ρ-VEX processor, the size of the ST200-STB1 is used. By correcting for the change in production process, 200 ρ-VEX processors can be placed on an accelerator. With a data transfer speedup of 200, the projected total speedup over the reference E8500 CPU is 10 to 15. In comparison to *graphics processing units (GPUs)*, the projected speedup is between 1.2 times faster and 8.8 times slower.

Due to architectural differences between an accelerator containing many ρ-VEX processors and a GPU, kernels that contain many branches are expected to achieve better performance on the ρ-VEX based accelerator. Further research is needed to determine applications with these type of kernels. Another area for future research is the architecture of the accelerator; it is expected that improvements to the cache and pipeline architecture can further improve the performance of the accelerator.

7.2 Main contributions

The problem statement of this thesis was:

In the OpenCL environment, can the ρ-VEX be competitively used as an accelerator?

To answer this question, this thesis had three main goals:

1. Designing and implementing a platform to use the ρ-VEX processor as an accelerator.

2. Determining the performance of the different components of the acceleration platform.

3. Creating and validate a model of the total execution time of a kernel.

To reach the first goal, an accelerator using the ρ-VEX processor has been implemented on the Xilinx ML605 platform. This accelerator communicates with the host using PCIe. Implementing OpenCL has been done by extending the pocl framework. A new Linux driver has been developed to support the communication between the pocl framework and the accelerator.

The performance of the ρ-VEX processor, communication system and OpenCL compiler have been evaluated. Based on these results a model has been created to predict the total execution time of the reference program. Separately, a comparison of the available VEX compilers has been made to determine the influence of the compiler on the execution time of a kernel.

Using the found model and compiler results, an analysis has been made of the performance of the system on a benchmark application. It was found that the implemented accelerator is two orders of magnitude slower than the reference platform. To answer the research question, a projection has been made of the performance of a ρ-VEX based accelerator when produced as an ASIC using modern techniques.

The following main contributions have been made:

- A PCIe interface has been made between the ρ-VEX running on an ML605 and the host PC.
- A Linux kernel driver has been created to interface with the ρ-VEX core from user space.
- The pocl framework has been extended to support the ρ-VEX accelerator.
- An analysis of the performance of the system has been performed.
- Projections have been made to determine the competitiveness of a ρ-VEX based accelerator.

Furthermore, some unrelated contributions to the ρ-VEX project have been made:

- An extension to the `rvsrv` debug tool has been created to allow debugging over PCIe.
- A CoSy based compiler for the ρ-VEX has been developed, see Appendix B.
- The LLVM back end for the ρ-VEX has been upgraded to LLVM 3.6.
- Implementations of several image related library functions in pocl have been added.

7.3 Future Work

The design of a ρ-VEX based accelerator offers multiple opportunities for future work. These suggestions for future work are categorized in implementation enhancements and research opportunities.

7.3.1 Implementation enhancements

The current system has only been tested with one ML605. While most of the interfaces have been designed to support multiple ρ-VEX accelerator devices, some might have been forgotten. More pressing is the lack of device locking: there is currently no mechanism with which a device can be reserved for use by only one program.

The current version of pocl does not execute queued kernel instances in parallel with the main program. It also does not support out-of-order execution of kernels on multiple cores. These features are required for a multi-core ρ-VEX accelerator to offer competitive performance compared to other OpenCL accelerators.

Not all features of OpenCL supported by pocl are implemented in the ρ-VEX back end. This includes filling a certain memory range with a specific value, copying data on the device, and mapping memory from the device to host memory. Operations as filling and copying of data can be performed on the ρ-VEX device itself, reducing the amount of PCIe communication.

The Linux kernel driver currently supports only one ρ-VEX accelerator. Adding support for more devices would require redesigning the `xdma` driver. Ideally, the `xdma` and `rvex` drivers are merged and a new device interface is automatically added when the correct PCIe ID is detected. The `rvex` driver has been designed to facilitate this merging.

There is currently no way for the kernel driver to query the size of the main memory on the ML605. Adding this information to the PCIe interface would allow having devices with differing main memory sizes.

7.3.2 Research opportunities

In Chapter 6 a very coarse projection has been made of the performance of an ASIC accelerator with ρ-VEX processors. A more in depth exploration of this design space is needed to provide more accurate projections. Additionally, a survey of kernels could be made that are branch-heavy and would thus benefit a ρ-VEX based accelerator.

On an accelerator with multiple cores, the efficient scheduling of kernel instances on these cores becomes important. As cores on the ρ-VEX can be rescheduled, the expected ILP of an instance can be used as a parameter for the scheduling. Furthermore, several different kernels might be available for execution at the same time. As instances of different kernel require different memory buffers to be available on the core, the transfer time of these buffers is also a parameter for the scheduler.

OpenCL specifies work-groups, kernel instances that have access to the same local memory and are modeled as running concurrently. In pocl, these instances are combined into a single instance. By adding small local memories to each core, and running the cores in a four times 2-issue configurations, running the work-group as four instances might lead to higher performance.

Bibliography

[1] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[2] (2015, November) Top500 List - June 2015. TOP500.org. [Online]. Available: http://www.top500.org/list/2015/06/

[3] (2015, November) Applications Using OpenCL. Khronos Group. [Online]. Available: https://www.khronos.org/opencl/resources/opencl-applications-using-opencl

[4] (2015, November) CUDA — Applications — GeForce. NVIDIA Corporation. [Online]. Available: http://www.geforce.com/hardware/technology/cuda/applications

[5] T. Van As, “ρ-VEX: A reconfigurable and extensible VLIW processor,” Master’s thesis, TU Delft, Delft University of Technology, 2008.

[6] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, “The Molen Media Processor: Design and Evaluation,” in *Proceedings of the International Workshop on Application Specific Processors, WASP 2005*, September 2005, pp. 26–33.

[7] F. Anjam, “Run-time Adaptable VLIW Processors: Resources, Performance, Power Consumption, and Reliability Trade-offs,” Ph.D. dissertation, 2013.

[8] R. Seedorf, F. Anjam, A. Brandon, and S. Wong, “Design of a Pipelined and Parameterized VLIW Processor: ρ-VEX v. 2.0,” in *HiPEAC Workshop on Reconfigurable Computing (WRC)*, 2012.

[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Elsevier, 2005.

[10] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, “Lx: A Technology Platform for Customizable VLIW Embedded Processing,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA ’00. New York, NY, USA: ACM, 2000, pp. 203–213. [Online]. Available: http://doi.acm.org/10.1145/339647.339682

[11] (2015, October) HP Labs : Downloads: VEX. Hewlett-Packard Development Company. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[12] (2015, October) STLinux. STMicroelectronics. [Online]. Available: http://stlinux.com/

[13] M. Daverveldt, “LLVM-based ρ-VEX compiler,” Master’s thesis, TU Delft, Delft University of Technology, 2014.

[14] *The OpenCL Specification, Version 1.2*, Khronos OpenCL Working Group, Specification, Rev. 19, November 2012. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[15] A. Wilen, J. Schade, and R. Thornburg, *Introduction to PCI Express.* Intel Press Santa Clara, 2003.

[16] (2015, November) Frequently Asked Questions — PCI-SIG. PCI-SIG. [Online]. Available: https://pcisig.com/faq?field_category_value[]=pci_express_3.0&keys=

[17] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[18] (2015, October) Beignet. [Online]. Available: http://www.freedesktop.org/wiki/Software/Beignet/

[19] (2015, October) GalliumCompute. [Online]. Available: http://dri.freedesktop.org/wiki/GalliumCompute/

[20] (2015, October) zuzuf/freeocl. [Online]. Available: https://github.com/zuzuf/freeocl

[21] (2015, October) pocl - Portable Computing Language. [Online]. Available: http://portablecl.org/

[22] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A Performance-Portable OpenCL Implementation," *International Journal of Parallel Programming*, pp. 1–34, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y

[23] M. Faddegon, "SSA Back-Translation: Faster Results with Edge Splitting and Post Optimization," Master's thesis, TU Delft, Delft University of Technology, 2011.

[24] (2015, October) DMA Back-End Core. Northwest Logic. [Online]. Available: http://nwlogic.com/products/docs/DMA_Back-End_Core.pdf

[25] (2015, October) Virtex-6 FGPA Connectivity Kit Documentation. Xilinx. [Online]. Available: http://www.xilinx.com/products/boards/v6conn/reference_designs.htm

[26] P. Mochel, "The sysfs Filesystem," in *Linux Symposium*, 2005, pp. 313–326.

[27] G. Kroah-Hartman, "udev–A Userspace Implementation of devfs," in *Proc. Linux Symposium.* Citeseer, 2003, pp. 263–271.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[29] J. Scott, L. H. Lee, J. Arends, and B. Moyer, “Designing the Low-Power M•CORE^{TM} Architecture,” in *Power Driven Microarchitecture Workshop.* Citeseer, 1998, pp. 145–150.

[30] *DDR3 SDRAM standard*, JEDEC, Std., Rev. C, November 2008.

[31] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, 1967, pp. 483–485.

[32] P. Faraboschi and F. Homewood, “ST200: A VLIW Srchitecture for Media-Oriented Applications,” in *Microprocessor Forum*, 2000, pp. 9–13.

[33] (2015, October) Intel Core2 Duo Processor E8500 (6M Cache, 3.16 GHz, 1333 MHz FSB) Specifications. Intel. [Online]. Available: http://ark.intel.com/products/33911/

[34] R. M. Hollander and P. V. Bolotoff. (2015, October) RAMspeed, a cache and memory benchmark. Alasir. [Online]. Available: http://alasir.com/software/ramspeed/

[35] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, “Evaluating Performance and Portability of OpenCL Programs,” in *The fifth international workshop on automatic performance tuning*, 2010, p. 7.

[36] (2015, October) List of AMD graphics processing units. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units

[37] (2015, October) Intel Core i7-920 Processor (8M Cache, 2.66 GHz, 4.80 GT/s Intel QPI) Specifications. Intel. [Online]. Available: http://ark.intel.com/products/37147/

[38] (2015, October) List of Nvidia graphics processing units. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

[39] (2015, October) GeForce 200 series. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/GeForce_200_series

[40] (2015, October) PassMark - CPU Performance Comparison. PassMark Software. [Online]. Available: http://www.cpubenchmark.net/compare.php?cmp[]=5&cmp[]=834

[41] M. Alt, U. Amann, and H. van Someren, “CoSy Compiler Phase Embedding with the CoSy Compiler Model,” in *Compiler Construction*, ser. Lecture Notes in Computer Science, P. Fritzson, Ed. Springer Berlin Heidelberg, 1994, vol. 786, pp. 278–293. [Online]. Available: http://dx.doi.org/10.1007/3-540-57877-3_19

A Kernel source code

```
#define imageWidth 640
#define imageHeight 480

static void convolution (const image2d_array_t image,
        image2d_array_t result, int index, constant int *filter,
        const int filter_size) {
  const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
    | CLK_ADDRESS_NONE | CLK_FILTER_NEAREST;

  for (int x = 0; x < imageWidth; x++) {
    for (int y = 0; y < imageHeight; y++) {
      int4 res_pixel = (int4)(0);

      for(int filterX = 0; filterX < filter_size; filterX++) {
        for(int filterY = 0; filterY < filter_size; filterY++) {
          int imageX = (x - filter_size / 2 + filterX + imageWidth)
              % imageWidth;
          int imageY = (y - filter_size / 2 + filterY + imageHeight)
              % imageHeight;
          const int4 coord = (int4)(imageX, imageY, index, 0);

          const int4 src_pixel = read_imagei(image, sampler, coord);
          res_pixel += (src_pixel * filter[filterX + filterY * filter_size])
              /256;
        }
      }

      //truncate values smaller than zero and larger than 255
      res_pixel = clamp(res_pixel, 0, 255);

      const int4 coord = (int4)(x, y, index, 0);
      write_imagei(result, coord, res_pixel);
    }
  }
}
```

Figure A.1: Original convolution kernel

```
#define imageWidth 640
#define imageHeight 480

static void convolution_opt (const image2d_array_t image,
        image2d_array_t result, int img_index, constant int *filter,
        const int filter_size) {
  const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
    | CLK_ADDRESS_NONE | CLK_FILTER_NEAREST;

  global dev_image_t* image_dev = *(global dev_image_t**)&image;
  global dev_image_t* result_dev = *(global dev_image_t**)&result;
  uchar4 *img = (uchar4*)image_dev->data;
  uchar4 *res = (uchar4*)result_dev->data;

  for (int x = 0; x < imageWidth; x++) {
    for (int y = 0; y < imageHeight; y++) {
      int4 res_pixel = (int4)(0);

#pragma unroll
      for(int filterX = 0; filterX < filter_size; filterX++) {
#pragma unroll
        for(int filterY = 0; filterY < filter_size; filterY++) {
          int imageX = (x - filter_size / 2 + filterX + imageWidth)
              % imageWidth;
          int imageY = (y - filter_size / 2 + filterY + imageHeight)
              % imageHeight;

          int pixel_index = imageX + imageY*imageWidth
              + img_index * imageWidth * imageHeight;
          int filter_index = filterX + filterY * filter_size;

          const int4 src_pixel = convert_int4(img[pixel_index]);
          res_pixel += (src_pixel * filter[filter_index])/256;
        }
      }

      int pixel_index = imageX + imageY*imageWidth
          + img_index * imageWidth * imageHeight;

      //truncate values smaller than zero and larger than 255
      res_pixel = clamp(res_pixel, 0, 255);
      res[pixel_index] = convert_uchar4(res_pixel);
    }
  }
}
```

Figure A.2: Optimized convolution kernel

CoSy compiler back end for VEX

B

CoSy is a compiler development system developed by ACE Associated Compiler Experts. It has an extendible intermediate representation and supports easy reordering and parallel execution of compiler phases[41]. The compiler phases, which are called engines in CoSy, are written or generated by the user of the development system. The code to manage data and start engines is generated by CoSy, based on specifications from the user on how and when the engines should be executed.

A VEX code generator will be developed for CoSy. This code generator is based on the sample code generator provided by CoSy.

B.1 Register File

The register file of the ρ-VEX consists of 64 general purpose registers of 32-bits, 8 branch registers and a link register. The general purpose registers can be used as source and destination for arithmetic operations. The branch registers are set by comparison instructions are consumed by branch instructions. The link register holds the address to return to after a function call completes.

The register definitions can be found in Figure B.1. It adds an extra set of registers: the double word registers (D3, D5, D7 and D9). These registers are composited of two general purpose registers and are used when passing 64-bit function arguments. Also note that R0 is declared as constant as it is always equal to 0.

Several register sets are defined that are used in different contexts (see Figure B.2). Note that R1 is not included in the set of general purpose registers, as it is used as the stack pointer. Two disjunct sets are defined for the registers that are saved by the caller and for those that are saved by the callee. Finally a rule is added to replace integer constants of value 0 by accesses to register R0.

B.2 Scheduling

Each instruction in CoSy is part of a consumer class and producer class. The producer class is propagated to the registers that are set by the instruction. The scheduler determines the minimum amount of cycles between two instructions based on the producer and consumer class. These values are provided in the upper part of Figure B.3.

The default producer (DEFPROD) is used for most instructions, its produced values can be consumed in the next cycle. Loads of general purpose registers and multiply instructions (LOADPROD and MULPROD) have a delay of one cycle, while link register loads (LDLPROD) have a delay of two cycles. Branch instructions have special delay

```
REGISTER
  // general-purpose registers
  FOR i = 0..63 R[i],

  // integer pair registers, used for function calling/returning
  FOR i = 0..3 D[2*i+3],

  //  branch registers
  FOR i = 0..7 B[i],

  // link register
  L0;

REGISTER COMPOSITION
  FOR i = 0..3 dbl:D[2*i+3] <hi:R[2*i+3], lo:R[2*i+4]>;

REGISTER CONSTANT
  // general purpose register 0 is always 0
  <R0>;
```

Figure B.1: Register definitions

```
REGISTER SET
  // Some register sets
  gp_regs             <R0,R2..R63>, // Don't include R1
  br_regs             <B0..B7>,
  dbl_regs            <D3,D5,D7,D9>,
  link_reg            <L0>,

  // see cgd/ccreg/target.c
  abi_callee_changed  <R2..R56, B0..B7, dbl_regs>,

  // see cgd/stacklayout/target.c
  abi_callee_saved    <R57..R63,L0>,

  // To make reads of the stack frame.
  readSP              <R1>,

  // Registers available for the register allocator
  avail               <R2..R63, br_regs, dbl_regs, L0>;
```

Figure B.2: Definition of register sets

requirements; demanding a one cycle delay between instructions setting a branch register and the branch itself.

To determine the amount of instructions that can be scheduled in the same cycle,

```
SCHEDULER
  PRODUCER
    DEFPROD, LOADPROD, LDLPROD, BREGPROD, MULPROD;
  CONSUMER
    DEFCONS, BRANCHCONS;

  TRUE      DEFCONS :
  DEFPROD   1,
  LOADPROD  2,
  LDLPROD   3,
  MULPROD   2;

  TRUE      BRANCHCONS:
  BREGPROD  2,
  MULPROD   2;

  RESOURCES
    alu<2>, mul<2>, ldst0, branch0, instr_width<2>;
  TEMPLATES
    DEFTMPL          := instr_width + alu + mul + ldst0 + branch0;
    ALUTMPL          := instr_width + alu;
    MULTMPL          := instr_width + mul;
    LDSTTMPL         := instr_width + ldst0;
    BRANCHTMPL       := instr_width + branch0;

    // Long immediates cost a syllable extra
    ALU_LIMM_TMPL    := instr_width + ALUTMPL;
    MUL_LIMM_TMPL    := instr_width + MULTMPL;
    // LDST long immediates are placed in the branch slot
    LDST_LIMM_TMPL   := instr_width + LDSTTMPL;
```

Figure B.3: Scheduler configuration

resource information is provided in the lower part of Figure B.3. These resources provide a model for the scheduler. The first four resources model the physical resources available in the core: Arithmetic Logic Units (ALU), MULtilply units (MUL), LoaD STore units (LDST) and branch units. The instruction width is also modeled as a resource.

Each instruction is part of a template, which defines the amount of resources the instruction uses. As the VEX instruction set allows the immediate arguments of instructions to have a 32-bit size, larger arguments do not fit in the 32-bit instruction itself. This is solved by storing the remainder of the instruction in an extra “long immediate” instruction, which does not perform any actions itself. While the generation of these long immediate instructions is performed by the assembler, the compiler has to take them into account while scheduling as there is otherwise no space in the bundle.

For instructions with an immediate value known by the compiler the situation is easy.

```
RULE [abs_int] o:mirAbs(rs:reg) -> rd:reg;
INTERFERE(rs, rd);
COST 10;
EXPAND {
  gcg_create_sub(gcg_expand, rd, RegR0, rs);
  gcg_create_max(gcg_expand, rd, rd, rs);
};
END
```

Figure B.4: `abs_int` rule lowering

By checking the size of the immediate value the instruction is assigned either its normal template or its long immediate template (suffixed by _LIMM_TEMPL). If the compiler does not know the value of the immediate, mostly when moving a label into a register, it errs on the side of caution and reserves an extra instruction slot.

B.3 Instruction lowering

Instructions are lowered by implementing rules. Each rule consumes an IR instruction and either expands into new instruction rules or directly emits assembly. Expanding into instruction rules is preferred, as this allows instruction based scheduling.

An example of a rule is shown in Figure B.4. This rule expands the mirAbs IR instruction into the VEX sub and max instructions. The INTERFERE property signals the compiler that the source register `rs` and the destination register `rd` can not be the same. The COST property assigns a weight to the rule, allowing the rule matcher to select the matches that have the lowest weight.

B.4 Calling convention

A calling convention of an architecture defines where arguments and return values are stored when a function is called. Arguments can be stored in registers or on the stack and the return value is often stored in a register. The VEX calling conventions are relatively straight forward: 32-bit or smaller arguments are stored in the registers R3 to R10, 64-bit arguments in the register combinations R3-R4, R5-R6, R7-R8 and R9-R10. Any arguments that don't fit are stored on the stack. The return value is stored in the R3 register when 32-bit or smaller, and otherwise in the registers R3 and R4.

Assignment of the calling convention registers in CoSy is handled by the `ccreg` engine. Then engine works by a set of conditional rules. When the condition of a rule is true and there is still a register free, that register is assigned as the location of the argument. Otherwise the next rule is tried.

The rules for the VEX calling convention are shown in Table B.1. First integer values of 32-bits or smaller are tried to be assigned. Then 64-bit integers, followed by pointers which are 32-bit in the VEX architecture. If none of these rules can be applied, the argument is stored on the stack.

Rule name	Condition	argument location
int	$\leq$ 32-bit integer	R3 ... R10
int64	64-bit integer	R3-R4, R5-R6, R7-8, R9-10
ptr	pointer	R3 ... R10
stack	true	stack

Table B.1: Calling convention rules

B.5 Stack layout

The stack layout of the VEX architecture is partly defined by the specification[9]. The important parts are that:

1. Outgoing arguments that do not fit in registers are placed in the Outargs region, which starts 16 bytes from the start of the frame.
2. A scratch area of 16 bytes is located at the start of the frame. This area can be used by a called method.
3. Stack pointers should be 32-byte aligned, requiring all stack frames to be a multiple of 32 bytes.

The stack layout as used by the CoSy compiler is shown in Figure B.5. From high to low addresses, the following things are stored on the stack:

- Variable argument storage: a 16 bytes area used to copy the R3 to R7 registers to when this method has variable argument length. When the method has a static argument length, this area has a size of 0 bytes.
- Link register slot: 4-byte area to store the link register. When this method calls another method, the link register will be overwritten. This slot can be used to store the link register so it can be restored before returning from this method call.
- Register save slots: area where registers can be saved. A method is allowed to change certain registers without restoring those, requiring the calling method to restore these registers after the call if required. Other registers need to be restored by the method itself. The register save slots are used to store these registers.
- Spill slots: When there are more registers required than there are physical registers, some registers are temporarily stored on the stack. This process is called spilling. The spill slots are used to store these spilled registers.
- Local objects: Object allocated on the stack are placed in the local object slot.
- Out arguments: Arguments to a method that do not fit in the registers are placed in this slot. The offset between the start of these arguments and the stack pointer is always fixed, so the called method knows where to find them.
- Scratch area: The scratch area can be used by the called method and is always 16 bytes long.

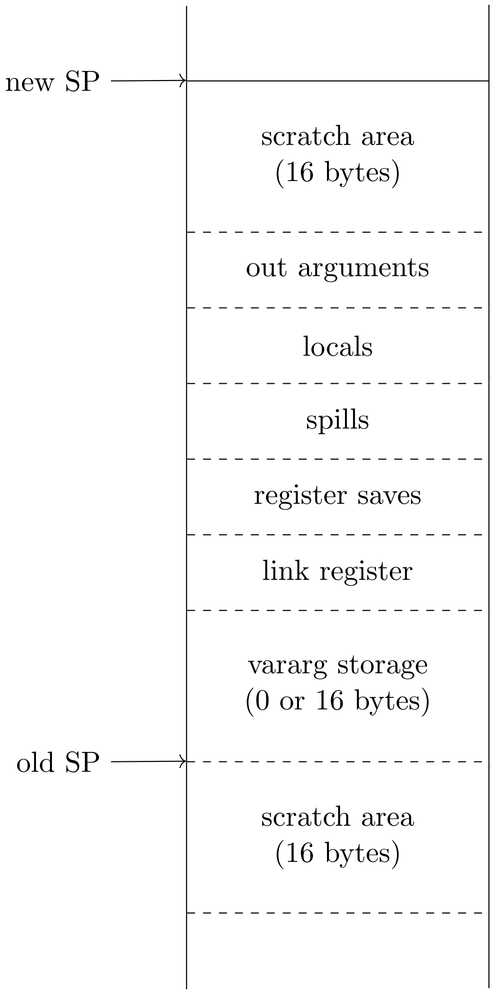

Figure B.5: Layout of a stack frame

The CoSy VEX generator does not use a frame pointer. It uses a fixed stack frame size with which the stack pointer is subtracted in the function prologue and which is added in the function epilogue. Adding frame pointers will be necessary to support functionality like the `alloca` function.

B.6 Variadic functions

A variadic function is a function in C that can have a variable amount of arguments. Probably the most well known is the `printf` function, with the signature `int printf(const char * format, ...)`. To access the arguments, the function uses the `va_start`, `va_arg` and `va_end` macro's. The `va_start` macro initializes a `va_list` argument to be used for argument list traversal. The next argument can be requested with the `va_arg` macro and the `va_end` macro cleans up resources used by `va_start`.

In the function prologue of a function with a variable argument list, the registers R3

```
#include <stdint.h>

typedef void *va_list;

#define __align(p, size) \
  (void*) ((size) == 1 ? (uintptr_t)(p) :  \
    ((size) == 2 ? (uintptr_t)(p) + 1 & ~0x1 : \
      ((size) <= 4 ? (uintptr_t)(p) + 3 & ~0x3 : \
        (uintptr_t)(p) + 7 & ~0x7 \
      ) \
    ) \
  )

extern void *__builtin_va_start(void);
#define va_start(ap, parmN) ((void) ((ap) = __builtin_va_start()))
#define va_arg(ap, type) \
  (* ( \
    (type *) ( \
      (ap) = (type *)__align(ap,sizeof(type)) + 1 \
    ) - 1 \
  ) )
#define va_copy(dst, src)   ((dst) = (src))
#define va_end(ap)          ((void) 0)

#undef __align
```

Figure B.6: Definitions of variadic functions in `stdarg.h`

to R10 are copied to stack in the variable argument storage slot and the scratch area slot of the previous stack frame. This results in a continues area of arguments, as the out-going arguments, scratch area and variable argument storage slots are placed next to each other. Reading the argument list is then implemented by incrementing a pointer that points to the location of the next argument.

The `va_start` macro is implemented as a built-in function. This function determines the offset of the start of the variable argument list, based on the function signature. All other macros are implemented in C, as shown in Figure B.6.

Biography

Hugo van der Wijst is a computer scientist with a strong interest in computer architecture, embedded software, and software project management. He started his studies in Computer Science at TU Delft in 2008, receiving his Bachelor's degree with Cum Laude distinction in 2012. In 2013 he continued his studies at TU Delft, working towards a Master's degree in Computer Engineering.

From 2009 to 2011, Hugo was a part-time engineer at Formula Student Team Delft. There, he worked on software for the data-acquisition system and the electronic control unit of both the last combustion and first full-electric race car of the team.

For the 2011-2012 competition year, he joined the board of the team as Chief Electronics. The car designed in that year reached second place at the Formula Student competition in Silverstone, England and became the overall winner of the Formula Student Electric competition in Hockenheim, Germany. In 2013, the car broke the Guinness World Record for fastest 0-100 km/s acceleration of an electric car.

In 2013, Hugo performed an internship at Tesla Motors in Palo Alto. He will return to Tesla Motors in a full-time position in 2016.