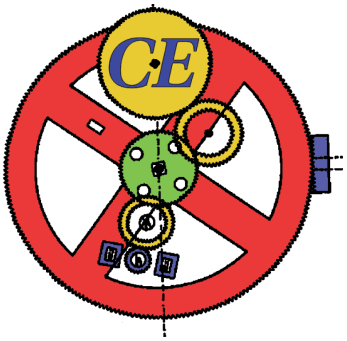

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

Exploring Convolutional Neural Networks on the ρ-VEX architecture

Jonathan Tetteroo

CE-MS-2018-10

Abstract

As machine learning algorithms play an ever increasing role in today's technology, more demands are placed on computational hardware to run these algorithms efficiently. In recent years, Convolutional Neural Networks (CNNs) have become an important part of machine learning applications in areas such as object recognition and detection. In this thesis we will explore how we can implement CNNs on the ρ-VEX processor and what can be done to optimize the performance.
The ρ-VEX processor is a VLIW processor that was developed at the Delft University of Technology and that can be reconfigured during runtime to take advantage of Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) in an application. In this work we have developed a streaming pipeline in a simulator consisting of multiple 8-issue ρ-VEX cores connected with memory buffers. This pipeline was designed to execute CNN inference and take advantage of the overlapped execution to increase throughput. Furthermore, as each ρ-VEX core can be configured to operate in a one core, two core or four core mode based on available ILP and TLP, we can adapt the processor based on the current operation being executed and the amount of parallelism that is available.
By generating the required code from a high-level description of the CNN, it becomes straightforward to test multiple configurations of the pipeline and determine which creates the best performance. The implementation was subsequently tested using a simple network trained on the MNIST dataset.
By dividing the workload of the convolutional layers over multiple contexts to take advantage of data-level parallelism, we improved the latency by 3.03× and the throughput by 3.14× in simulation. By creating a pipeline of six cores in a single context configuration in the simulator, we achieved a throughput increase of 1.77×. A hardware implementation of the pipeline was also synthesized for a Virtex-6 FPGA, consisting of four 2-issue ρ-VEX cores at a clock speed of 200 MHz. We subsequently propose several optimizations to increase performance of CNN inference on the ρ-VEX architecture.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Exploring Convolutional Neural Networks on the ρ-VEX architecture

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jonathan Tetteroo
born in Mission Viejo, United States

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

Exploring Convolutional Neural Networks on the ρ-VEX architecture

by Jonathan Tetteroo

Abstract

As machine learning algorithms play an ever increasing role in today's technology, more demands are placed on computational hardware to run these algorithms efficiently. In recent years, Convolutional Neural Networks (CNNs) have become an important part of machine learning applications in areas such as object recognition and detection. In this thesis we will explore how we can implement CNNs on the ρ-VEX processor and what can be done to optimize the performance.

The ρ-VEX processor is a VLIW processor that was developed at the Delft University of Technology and that can be reconfigured during runtime to take advantage of Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) in an application. In this work we have developed a streaming pipeline in a simulator consisting of multiple 8-issue ρ-VEX cores connected with memory buffers. This pipeline was designed to execute CNN inference and take advantage of the overlapped execution to increase throughput. Furthermore, as each ρ-VEX core can be configured to operate in a one core, two core or four core mode based on available ILP and TLP, we can adapt the processor based on the current operation being executed and the amount of parallelism that is available.

By generating the required code from a high-level description of the CNN, it becomes straightforward to test multiple configurations of the pipeline and determine which creates the best performance. The implementation was subsequently tested using a simple network trained on the MNIST dataset.

By dividing the workload of the convolutional layers over multiple contexts to take advantage of data-level parallelism, we improved the latency by 3.03× and the throughput by 3.14× in simulation. By creating a pipeline of six cores in a single context configuration in the simulator, we achieved a throughput increase of 1.77×. A hardware implementation of the pipeline was also synthesized for a Virtex-6 FPGA, consisting of four 2-issue ρ-VEX cores at a clock speed of 200 MHz. We subsequently propose several optimizations to increase performance of CNN inference on the ρ-VEX architecture.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-10

Committee Members :

Advisor: Stephan Wong, CE, TU Delft

Chairperson: Stephan Wong, CE, TU Delft

Member: Arjan van Genderen, CE, TU Delft

Member: Jan van Gemert, INSY, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ALU Arithmetic Logic Unit

ANN Artificial Neural Network

ASIC Application Specific Integrated Circuit

BLAS Basic Linear Algebra Subprograms

BRAM Block RAM

BVLC Berkeley Vision and Learning Center

CNN Convolutional Neural Network

CPU Central Processing Unit

DAG Directed Acyclic Graph

DDR Double Data Rate

DLP Data Level Parallelism

DMA Direct Memory Access

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

DSP Digital Signal Processing

FIFO First-In-First-Out

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

ILP Instruction Level Parallelism

IP Intellectual Property

LUT Look-up Table

MAC Multiply-Accumulate

MNIST Modified NIST Database

MSB Most Significant Bit

NIST National Institute of Standards and Technology

PE Processing Element

PPC PowerPC

PRF Polymorphic Register File

RAM Random Access Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SPU Synergistic Processing Unit

TLP Thread Level Parallelism

TPU Tensor Processing Unit

VEX VLIW Example

VGA Video Graphics Array

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

Acknowledgements

First of all, I would like to thank Stephan Wong for his supervision and helping me with ideas. I would also like to thank Joost Hoozemans and Jeroen van Straten for their technical assistance and advice. I also thank Arjan van Genderen and Jan van Gemert for being on my graduation committee. And finally, I would like to thank my parents for their support over the years.

Jonathan Tetteroo
Delft, The Netherlands
March 27, 2018

1 Introduction

In this thesis we explore how to implement Convolutional Neural Networks (CNNs) on the ρ-VEX architecture. As machine learning becomes more important in everyday life, in particular related to internet applications, there is more demand for efficient solutions for implementing these types of algorithms. CNNs are part of the broader field of Deep Neural Networks (DNNs) which have become the driving force behind technologies such as image classification [10] and speech recognition [11].

To train and execute these networks effectively, high powered CPUs and GPUs are commonly used [12]. However, as these types of applications move out of the data center and onto smart devices, other solutions may need to be found to run these networks efficiently due to limitations on energy usage and computational performance [13].

We investigate how the ρ-VEX processor can be used to run CNNs and propose an architecture to run these types of networks efficiently.

In this chapter we present an introduction to the problem and the context in which it is set. This is followed by the problem statement and goals for the thesis. Finally, a brief overview of the rest of the chapters is given.

1.1 Context

Machine learning systems are playing an increasing role in the connected society of today. Many important internet applications rely on machine learning, such as web searches and e-commerce recommendations, as well as ubiquitous consumer devices such as cameras and smartphones. The machine learning systems are used for image recognition, speech recognition and selecting products and web results based on a user's interests [4].

Machine learning was defined in 1959 by Arthur Samuel as the field of study that gives computers the ability to learn without being explicitly programmed and encompasses research from artificial intelligence, probability and statistics, neurobiology and several other fields [14].

A common form of machine learning is supervised learning. In supervised learning an algorithm is trained based on a training data set, for example for recognizing cars and boats in photographs. The algorithm can learn to recognize cars or boats by using this training dataset and then perform this task on arbitrary photographs providing the algorithm is generalizable enough. These types of tasks are known as classification tasks because they classify the input into a certain category.

One such implementation of machine learning is through the use of artificial neural networks. An artificial neural network is loosely based on the structure of the nervous system found in animals and humans and revolves around the idea that a massive network of simple parallel processors connected through weighted links can approximate

complex nonlinear functions. By placing multiple layers of artificial neurons in series and determining the link weights by means of a learning algorithm, an accurate approximation can be found. When applying a trained artificial neural network to a task such as image classification of arbitrary photographs, this is referred to as inference.

To approximate more complex behaviors, such as in the case of image classification, deeper networks can be used that consist of many layers of neurons. This is called Deep Learning or Deep Neural Networks (DNNs) and has become an important part of recent machine learning research. However, because of the increased network complexity, training and executing the networks becomes computationally expensive.

To reduce the complexity of the network, CNNs provide a solution by considering the input data as a structured grid. By performing convolutions over the input data using small weight kernels, the number of weights can be reduced because we use a single kernel across the entire input map instead of a weight for each data point of the input map. The CNN then also creates a successively higher abstracted representation of the input data by reducing extracted input features to single activations in subsequent layers. This also decreases computational complexity and makes it tractable to train highly complex CNNs on GPU platforms for applications such as image classification.

GPUs are a good candidate as accelerator for training these types of networks because of the highly parallelizable nature of Artificial Neural Networks (ANNs) and CNNs. These calculations mainly consist of performing multiply-accumulate operations in parallel and then sending the result through nonlinear activation functions. By mapping these operations to matrix multiplications, these can be calculated with high performance in GPUs at the cost of some data redundancy in the matrices [15]. However, GPUs are designed to accelerate primarily graphics applications and not specifically neural networks, which means a dedicated accelerator tailored towards the specific data structures of neural networks can achieve better performance per watt.

To improve this situation, special architectural features have been implemented in CPUs and GPUs to increase the performance of deep learning applications. For example, the NVIDIA PASCAL P100 architecture has support for performing two 16-bit floating point operations simultaneously in a single precision core to improve deep learning performance [16], while the Intel Knights Landing CPU contains special deep learning vector instructions [17].

To improve performance and increase energy efficiency, several dedicated accelerators for deep learning and convolutional neural networks have been developed in the last few years. Examples of research accelerators are neuFlow [18] and DianNao [19]. An example of an accelerator for deep learning applications used in industry is Google's Tensor Processing Unit (TPU) [20].

These accelerators recognize the fact that the main bottleneck for DNNs is the memory access, as a Multiply-Accumulate (MAC) operation requires three memory reads and one memory write. It means that in the worst-case scenario, all the memory accesses have to go through the external DRAM which negatively affects performance and energy usage. The accelerators generally solve this by introducing multiple levels of local memory to reduce data movement and thus energy usage.

However, dedicated accelerators are not a possibility in every scenario. In embedded systems and low-power systems the only available resource might be an embedded CPU

or a reconfigurable logic unit such as an Field Programmable Gate Array (FPGA). In these cases, the available resources have to be used in a way that a deep learning application, such as a CNN, can be executed in the most efficient manner.

In embedded CPUs, the available amount of computing power is generally limited, as well as lacking functionality such as floating point computation. As CNN computation is usually floating point based, in this case the network would have to be converted to fixed point computation [21]. Because of the limited computational power, the network generally also needs to be simplified to reduce the computational complexity, which can be achieved by means of network pruning and compression techniques [14] at the cost of performance or accuracy.

Because these CPUs are designed for general purpose applications, their performance will not match that of a dedicated accelerator. An alternative would be a platform with reconfigurable logic such as an FPGA, which can maintain its flexibility with regard to applications, while being able to optimize the performance by changing the way the application is implemented in reconfigurable logic.

When considering an FPGA platform, a suitable mapping of a CNN to the FPGA logic fabric needs to be found. Several examples of such mappings have been developed that consist of an array of Processing Elements (PEs) together with on-chip buffer memories, to achieve high throughput while being energy efficient [22].

By taking into account the available FPGA resources, design exploration can be performed to maximize the throughput given a certain CNN and FPGA type [23] [24]. Unfortunately, because these PE arrays are tailored towards DNN type problems, other types of applications cannot be executed on the FPGA. If we want run multiple applications without dividing up the FPGA, it will need to be reprogrammed every time, which might not be possible given a certain platform.

To avoid this problem, a softcore microprocessor such as the Xilinx MicroBlaze [25] can be implemented on the FPGA to handle general purpose applications but this leads us back to the limitations of the general purpose CPU with regard to performance. Ideally, we would require a CPU that can adapt to the current workload in order to maximize the performance, while still being able to execute every possible task.

We consider the ρ-VEX processor for this purpose. The ρ-VEX processor is a reconfigurable Very Long Instruction Word (VLIW) processor developed at the Computer Engineering department of the Delft University of Technology [26], that can adapt to a given workload by exploiting changes in Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP) by reconfiguring itself during runtime.

It can change from a single context 8-issue processor suited for high ILP applications down to a four context 2-issue processor for high TLP applications, among other configurations. The term context refers to a single execution context, which can be viewed as a single core, therefore by having multiple contexts running in parallel we effectively create a multi-core processor. It has been shown [27] that the ρ-VEX softcore on an FPGA can outperform the MicroBlaze softcore up to a factor of 3.2 for image processing applications by utilizing ILP efficiently.

When using the ρ-VEX to run CNN inference, we can adapt the processor to the current layer of the network that is being executed to exploit either the ILP or TLP present in that specific layer implementation to improve performance. However, we are

limited to a single ρ-VEX processor, so we can only improve the performance to a degree. Therefore, to improve performance further, a multi-processor ρ-VEX configuration could provide a solution. We will explore such an architecture in this thesis.

1.2 Problem statement and thesis goals

In [24] the CNN is mapped to a synchronous dataflow [28] representation in such a way that it can be mapped to an FPGA as a pipeline. In [7] a streaming fabric for medical imaging applications is proposed, that utilizes single 2-issue ρ-VEX cores as smallest processing elements. This fabric connects multiple cores together in a stream in such a way that pipelining can be exploited. We propose to connect these two ideas by creating a pipeline of full ρ-VEX cores for CNN inference. This leads to the following research question:

How can the ρ-VEX processor be utilized to perform CNN inference and how can performance be improved?

The main goals for this thesis are:

- Investigate how CNNs can be evaluated on a ρ-VEX processor platform.
- Implement a basic software implementation of a CNN network that can run both in a simulator as well as on a ρ-VEX hardware platform as a proof of concept.
- Implement an automated code generation application that generates all the required code from a high-level description to execute a CNN on the ρ-VEX processor to aid with quick design exploration.
- Find an efficient way to improve the inference performance of the ρ-VEX processor.
- Create a design that implements these efficiency improvements.
- Implement the design in the simulator and in hardware.
- Measure the performance and correctness of the CNN inference and evaluate the results.
- Provide further possible optimizations.

1.3 Contributions

The main contributions of this thesis are as follows:

- Created a proof of concept implementation of CNN inference on the ρ-VEX platform, tested both in simulator and hardware. It was verified to match the reference network on the entire MNIST [29] test dataset.

- Added multi-core support, streaming memories and configuration file loading to the ρ-VEX simulator. The code of the simulator was reorganized in such a way that all processor state is contained in a single data structure. The number of cores and its configuration parameters can now be passed through a configuration file.
- Designed a streaming pipeline for CNN inference based around reconfigurable 8-issue ρ-VEX cores and tested this in a simulator.
- Created a code generation tool that can generate the required code for CNN inference on the streaming pipeline based on a high-level description file. A subset of CNN layers was implemented as building blocks to facilitate this.
- Adapted the ρ-VEX streaming fabric hardware description to enable execution of our CNN inference code by increasing size of instruction and data memories and adapting the debug bus.
- Synthesized a four core 2-issue ρ-VEX stream with memories sized for CNN inference of the Modified NIST Database (MNIST) dataset at a clock speed of 200 Mhz.
- Investigated different improvements that can be made to increase the performance of CNN inference on the ρ-VEX platform.

1.4 Thesis outline

The thesis is organized in the following way:

Chapter 2 introduces more background information about parallelism, the ρ-VEX architecture, the benefits of dynamic reconfigurability and the available platform implementations. We then explore machine learning and artificial neural networks, before discussing convolutional neural networks in more detail. Subsequently, we review some of the challenges of implementing a CNN on an embedded system and current developments in that area.

Chapter 3 discusses how the CNN can be modeled as a streaming application and how such an application can be mapped to the ρ-VEX platform from a high-level description. We then investigate how we can take advantage of the features of the ρ-VEX to improve the performance of the application.

In *Chapter 4* the implementation of the code generation and the platform is discussed. The various components and utilized technologies are reviewed and there is an overview of what work has been done to achieve our goals.

Chapter 5 will evaluate the correctness of the implementation as well as the performance achieved. Based on these results we will investigate any further possible optimizations that could improve the performance and to what degree.

Finally, in *Chapter 6* a summary will be given of the work that has been done and of the results that were achieved. The main contributions and possible future work are also listed.

2 Background

In this chapter the background information required for the thesis will be discussed. First, we review several types of parallelism that are observed in computer applications. Subsequently, the ρ-VEX architecture and its implementation details are reviewed, especially the reconfiguration features and how this benefits performance with regard to taking advantage of parallelism. After this we introduce machine learning and we review in more detail artificial neural networks and convolutional neural networks. Finally, we examine how convolutional neural network inference is computed in practice and what requirements have to be met to perform this efficiently.

2.1 Computational parallelism

To improve performance of computation, several kinds of parallelism that exist in applications can be exploited. Parallelism implies there are tasks that can be performed simultaneously because they have no dependencies on each other, the amount of computations per cycle can then be increased by means of simultaneous execution. We analyze three different types of parallelism that can appear [30]:

Instruction Level Parallelism (ILP)

By using pipelining in processors, the execution of instructions can be overlapped which enables parallel execution of instructions and increases performance. The amount of ILP that can be extracted from instructions depends on the data dependencies and branches present in the instruction stream. The less dependencies there are, the more instructions can be executed in parallel and the higher the ILP will be. There are generally two approaches to take advantage of the ILP in a processor architecture: (dynamic) superscalar architectures or VLIW architectures.

ILP can be further improved by software techniques such as loop unrolling, this unrolls several iterations of a loop, removing some of the branch instructions. If the loop iterations are data-independent, the instructions of different iterations can be scheduled together to increase the ILP.

Thread Level Parallelism (TLP)

When application instructions have low ILP due to dependencies, it becomes difficult to extract more performance out of a single core processor that is optimized for exploiting ILP. To improve the performance, the application can be split into several independent tasks that run on execution threads. These threads can then each be executed in parallel on multiple cores to improve the performance, but requires the application to be suitable

to be split into multiple independent tasks. Because the threads operate in parallel, any dependencies between the threads need to be carefully managed with synchronization. Multithreading can increase the performance of certain applications significantly but places demands on the programmer to efficiently manage the different threads and avoid problems such as deadlocks and race conditions.

Data Level Parallelism (DLP)

If an application performs the same computations on a set of data that is independent, there is data level parallelism available that can be exploited. When running a multiprocessor system, each processor can perform the same function on a different set of data in parallel and improve the performance. Note the similarity with thread level parallelism, however in this case the data is different but the task is the same for each core. For single core operation, data level parallelism can also be exploited with the Single Instruction Multiple Data (SIMD) principle in which a single instruction operates on multiple data elements simultaneously.

An example of a SIMD system is a vector processor, which has a large sequential register file to store multiple data elements and operates on all data elements simultaneously before storing them back into memory. By pipelining the memory operations efficiently, the vector processor can reduce the initial setup latency for loading a vector to the same initial latency as loading a single scalar value.

2.2 The ρ-VEX architecture

Now that we understand the different types of parallelism present in applications, we can try to find a way to take advantage of these. One option is to use the ρ-VEX architecture that was designed with optimal TLP and ILP performance in mind and which we will review in this section. First, we analyze the reasoning behind reconfigurable computing and VLIW architectures. Subsequently, the design and features of the ρ-VEX processor are explained.

2.2.1 Reconfigurable computing

When observing computations inside electronics or computers, these are performed using either hardware or software. Computation in hardware can be performed by Application Specific Integrated Circuits (ASICs) that are highly efficient at the task they were designed for, but can unfortunately only perform one or a small set of tasks well. They also require a large economical investment with regard to design and fabrication and therefore are only feasible for mass market applications or specialized fields such as aerospace. Computer software on the other hand can execute varying tasks on general purpose hardware, which can be mass-produced at a reasonable cost. The drawback is that the performance is orders of magnitude less than a dedicated hardware implementation.

To bridge this gap, we look towards reconfigurable computing as a solution and generally we think of FPGA platforms in this regard. In an FPGA the logic is configurable

compared to fixed in an ASIC. By reprogramming the connections between the logic blocks inside the FPGA, complex circuits can be formed that can run computations in a more efficient way than a general purpose processor that runs software [31, p. xxv].

However, because it still relies on small general purpose logic blocks, the performance does not match that of an ASIC implementation. Furthermore, reprogramming the FPGA continuously is not a straightforward task. Quickly adapting to changes in either the application or computational workload is therefore difficult and for this scenario a better solution has to be found. The ρ-VEX can provide an answer and occupy the space between full FPGA reconfiguration and a general purpose processor. This is achieved by reconfiguring the hardware during runtime to minimize the reconfiguration delay, while still maintaining enough flexibility to extract more performance out of the software.

2.2.2 VLIW architecture

If we have an application with high ILP to take advantage of, we generally look at either superscalar or VLIW architectures. As the ρ-VEX processor is based around a VLIW architecture, we will not consider superscalar architectures in this study.

In VLIW architectures the processor has multiple independent functional units that can execute an instruction. By packaging multiple instructions into a single long instruction word, these instructions can be executed in parallel in a single cycle to take advantage of the ILP and improve performance. The amount of instructions in an instruction word or packet is called the issue-width of the processor and is equal to the maximum number of instructions per cycle.

The instructions are arranged into these long words by the compiler and thus the code is statically scheduled. It moves the burden of scheduling the instructions to the compiler side and only has to be performed once, which will save both energy and logic area in the processor. The drawback is that code compiled for a certain VLIW issue-width cannot run on a different issue-width directly and will have to be recompiled. A VLIW also places more demands on the compiler to schedule the instructions in an efficient way. In practice the full issue-width cannot always be used due to dependencies that exist between instructions and it is up to the compiler to schedule these as efficiently as possible with the aid of scheduling algorithms.

2.2.3 The ρ-VEX processor

After discussing reconfigurable computing and VLIW architectures, we understand the rationale and principles behind the ρ-VEX processor and will now look in more detail at its implementation.

The ρ-VEX is a VLIW processor based on the HP VLIW Example (VEX) family architecture [32] and was developed at the Computer Engineering department of the Delft University of Technology [33] [34] [26] .

The ρ-VEX consists of eight execution lanes called pipelanes in the default configuration, each of which can execute an instruction in parallel with the other lanes according to the VLIW principle (see Figure 2.1). The fetched instruction packet will contain a number of 32-bit instructions equal to the number of lanes. In this way the ILP of an

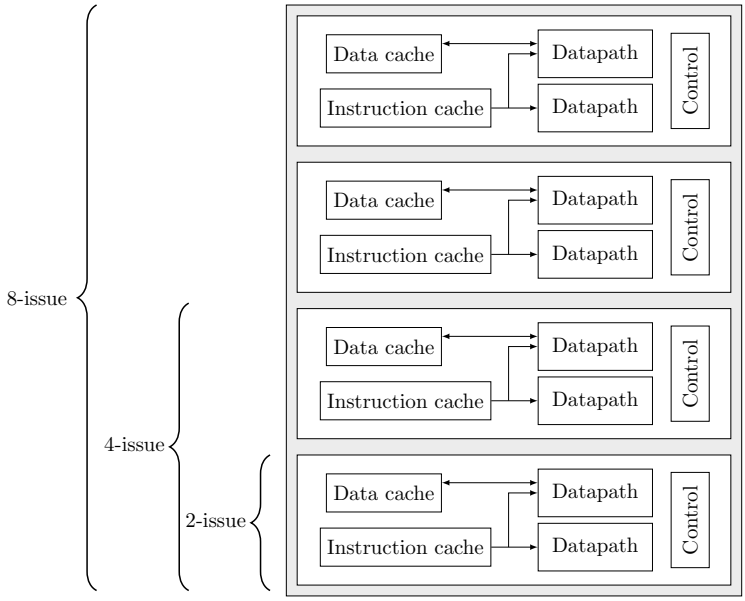

Figure 2.1: Simplified illustration of an 8-issue ρ-VEX core (adapted from [1])

application can be exploited by issuing multiple instructions per cycle and executing them in parallel.

To be able to exploit TLP there are four execution contexts, each with its own register file. Each context can be mapped to a subset of execution lanes to act as a single VLIW core. For example, there can be two execution contexts mapped to four lanes each, creating two independent 4-issue VLIW cores. The pipelanes are grouped in pairs, therefore the minimum size of a context is two lanes, which is a requirement for the ρ-VEX long-immediate instructions. In the default configuration each pipelane pair has one load/store memory unit and both have a multiplier and Arithmetic Logic Unit (ALU), in order to run any application on the minimum of two lanes.

Memory architecture

The ρ-VEX processor connects to memory through the memory unit in the pipelanes. When the ρ-VEX is configured in four 2-issue cores, each core can execute one memory instruction at a time. If multiple lane pairs are combined, only one memory unit can be active per context because the current data cache only supports one operation per cycle.

Caches

Cache memories are necessary to prevent the latency of off-chip memories to dominate the performance of the ρ-VEX processor. The ρ-VEX processor accesses instructions and data in parallel, which means the caches are split into a separate data cache and instruction cache. Each ρ-VEX execution lane fetches a 32-bit instruction per cycle and if the execution lanes are combined into a single core, their fetch ports work together to fetch a single aligned block of memory.

The ρ-VEX data cache can access one memory location per cycle for each lane pair. It can read 32-bit values or write 8-bit, 16-bit or 32-bit values. If lanes are combined into a single core, multiple memory operations can be executed simultaneously.

Each lane pair has its own caches that can be accessed independently from the other lanes. If lanes are combined into a single core, the caches are also combined with a reconfigurable interconnect.

Designtime and runtime (re)configuration

This processor can be configured during both design time and runtime in multiple ways. The key benefit is that you can change features such as issue-width and computation units without having to modify the processor manually. For example, each execution lane can be configured to have a multiplication unit to optimize the processor for multiplication intensive applications. If it is necessary to save resources, less multipliers can be configured even though the same application could still run on the processor with reduced performance.

The ρ-VEX processor can be configured at design time by adapting VHDL configuration files. Features that can be changed are the issue-width, available functional units and cache sizes. It gives the ρ-VEX design flexibility and it can be adjusted based on both the requirements of the application and the available hardware resources.

The runtime reconfiguration features of the ρ-VEX revolve around changing the issue-width of the processor. The current generation of the ρ-VEX processor can be configured as a single 8-issue context, two 4-issue contexts or four 2-issue contexts. It is also possible to have one core in 4-issue mode and the other two in 2-issue mode, giving a total of three cores (see Figure 2.2). This can be configured at runtime by preserving the register file of each execution context. By preserving the register file, execution can continue directly after reconfiguration which minimizes the delay to only a few of cycles.

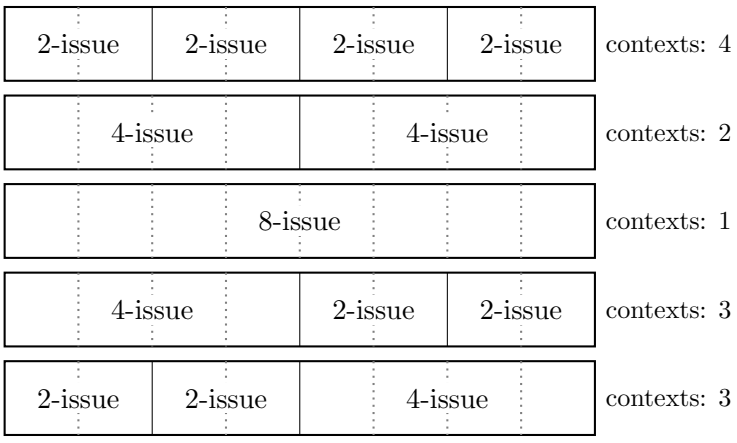

Figure 2.2: Possible core configurations with number of contexts, dotted lines indicate individual pipelanes.

The practical benefit of this reconfiguration is that the processor can assign all available execution lanes to a single thread if the application has high ILP, or split it into two

or more contexts if the application has higher TLP. This can be done at a fine-grained level in time because the reconfiguration delay is small. Contexts can also be scaled down to a smaller issue-width or shut down completely, which both could be used as a way to reduce energy consumption [35].

Simulator

A simulator of the ρ-VEX platform has been developed to test application code without the tight constraints on memory size and hardware resources. It is a modified version of the st200 simulator that targets the ST200 VLIW microprocessor. The simulator is written in the C programming language and supports all the reconfiguration features of the ρ-VEX processor and the full ρ-VEX instruction set. This simulator can easily be modified to implement new hardware features such as memory interfaces and peripherals.

Generic binary

The ρ-VEX processor can adapt to changes in ILP in an application by changing the issue-width of the processor, however the code binary would have to be recompiled to be able to run the same application code at a different issue-width. To solve this, a generic binary format for the ρ-VEX was developed in [36].

When compiling code as a generic binary, it can run at any issue-width that the processor is currently configured for. In this way the processor can reconfigure the issue-width in runtime while keeping the same code running, instead of requiring a specific binary for a certain issue-width. It comes with a performance penalty of between 1.3× and 1.09× compared to issue-width specific compilation.

ρ-VEX toolchain

A dedicated toolchain was developed for the ρ-VEX processor consisting of a compiler, assembler and standard libraries to support application development. The compiler/assembler is based on the Open64 compiler toolchain for VLIW architectures and can generate the ρ-VEX generic binary format. The Newlib [37] library is used as a standard C library, which is intended for embedded systems.

FPGA implementations

The following are the two main platforms for the ρ-VEX to run on FPGA development boards:

- A standalone processing system that utilizes only local memories implemented as FPGA Block RAMs (BRAMs). It also supports configurable cache memories with customizable latencies to simulate realistic caches.
- A GRLIB [38] based system where each lane group has a memory bus interface and interrupt support. GRLIB is a set of Intellectual Property (IP) cores written in VHDL that have been designed for System-On-Chip development. This platform supports off-chip DDR memory by means of the GRLIB DDR memory interface.

Other platforms exist but are not relevant to this thesis.

ρ-VEX streaming computation fabric

In [7] an FPGA computation fabric based on 2-issue ρ-VEX cores is introduced that uses a streaming architecture to accelerate image processing algorithms on FPGAs. This is achieved by performing each step of the algorithm on a single core and pipelining these cores in a stream configuration to increase the throughput. Multiple streaming pipelines can then be placed in parallel in the FPGA to each process a part of the input image and exploit data-level parallelism. Intermediate data in the stream is stored in local BRAM memories to reduce latency and the input and output of the streams are connected to Direct Memory Access (DMA) interfaces to move data to and from a host system.

2.3 Machine Learning

Having discussed parallelism and the ρ-VEX architecture, we now review the required knowledge for implementing convolutional neural networks, starting with machine learning.

Machine learning algorithms are useful for solving problems that cannot easily be solved by programs or algorithms that were manually designed by humans. Examples of such problems are: learning to recognize spoken words, driving an autonomous vehicle and classifying astronomical data.

The key characteristic of machine learning is that the computer trains itself to solve a problem, either by comparing results to an existing result set or by adapting results using a mathematical cost function. The machine takes away the task of solving the problem from humans and develops its own algorithm and method for finding solutions.

Machine learning is a multidisciplinary field that encompasses research from artificial intelligence, probability and statistics, neurobiology and other fields [39, p. 2]. It applies to a broad set of tasks and techniques and in this thesis we will focus on solving classification problems with the help of convolutional neural networks.

2.3.1 Supervised and unsupervised learning

Machine learning approaches can be roughly divided into two categories, supervised and unsupervised learning. In supervised learning an algorithm is trained with a training data set, the desired outcome of which is known. In this way the output of the algorithm can be compared to the known result. By calculating the error between the output and the correct result, the algorithm can be adjusted to yield better results.

Given a sufficiently large training set and the correct learning model, the algorithm can be generalized enough to function on any real world data with reasonable accuracy. One may, for instance, think of training an image classifier to recognize cars in photographs. By feeding into a machine learning model a large number of photographs with and without cars and also indicating which photographs have cars in them, the algorithm can be trained to eventually independently recognize cars in any photographs. Because the algorithm is explicitly informed which photographs contain cars during training, this

is called supervised learning. Convolutional neural networks are commonly a part of a supervised learning system for image recognition.

An unsupervised learning algorithm can determine useful features from a dataset independently without external help. For example, a dataset could be automatically clustered into several groups of similar samples without necessarily knowing the exact type of data.

We will not consider unsupervised learning further in this study.

2.3.2 Classification problems

A common task that can be solved with machine learning is classification problems. In this task, the computer is asked to determine in which of k categories a certain input belongs. The input can be anything, such as a photograph of an object or an audio sample. The algorithm produces a function $f : R_n \rightarrow \{1, .., k\}$ that takes the input vector x and produces an output value y $(y = f(x))$,where y is the numeric code of the specific category to which the input vector belongs.

A concrete example of this would be object recognition, where a photograph of an object is the input and the output is a numeric value indicating the type of object. It is also possible for the function to output a probability vector of the categories, to indicate which category the input is most likely to belong to, this is illustrated in Figure 2.3.

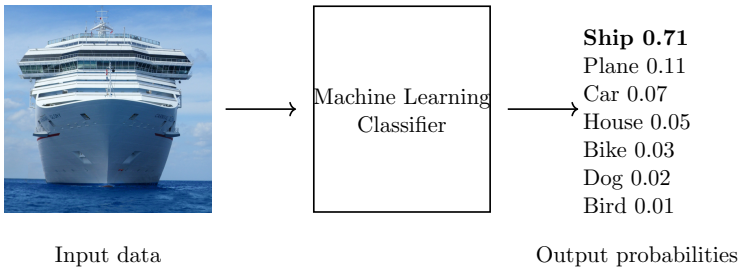

Figure 2.3: Basic image classifier with confidence scores

2.4 Artificial Neural Networks

After discussing machine learning and classification, we now investigate how this can be implemented in practice. Implementing machine learning algorithms can be done in several ways, one of which is using artificial neural networks. When talking about artificial neural networks, we usually refer to a nonlinear system with a structure that is loosely based on the nervous system found in animals and humans.

The neuron is the primary component in the human brain and consists of a few major parts: a dendrite that collects signals from other neurons and the cell body that processes the signals from the dendrite and generates an appropriate response which is sent along the axon. The axon branches out, connecting to other neurons through synapses (see Figure 2.4). The output response of a neuron is a nonlinear function of

its inputs and internal state and these neurons can generate intricate behaviors when connected in massive interconnected systems.

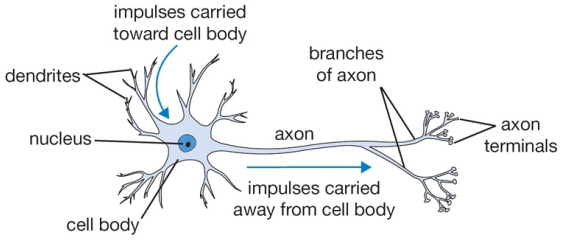

Figure 2.4: A schematic representation of a biological neuron (from [2])

Real world biological neurons are more complex than this, but artificial neural networks borrow from this idea by creating a massive network of simple processors that can operate in parallel and are interconnected through weighted links. These processors are then called neurons to remain with the analogy.

We view the neuron as an information-processing unit that is part of a neural network. The combination and interconnection of these neurons is the key to the correct operation of the network. In each neuron there are four elements to be considered, which are also illustrated in Figure 2.5:

- A set of synapses or connections that each have a specific weight assigned to them. For example a signal x_j is multiplied with a weight w_{kj} for synapse j and neuron k.
- An addition unit that sums the weighted input signals as a linear combination.
- An externally applied bias value b_k that decreases or increases the output value of the addition by a constant term.
- A nonlinear activation function that takes the output of the addition and limits the amplitude output of the neuron. For this reason it is also called a squashing function because it limits the permitted output amplitude range. Typically the output is either contained within the intervals $[0, 1]$ or $[-1, 1]$.

The neuron can be described by equations 2.1 and 2.2.

$$u_k = b_k + \sum_{j=1}^{m} w_{kj} x_j \tag{2.1}$$

$$y_k = \phi(u_k) \tag{2.2}$$

The input signals and weights are $x_1, ..., x_m$ and $w_{k1}, ..., w_{km}$ respectively. The output of the addition is u_k and b_k is the external bias value. Finally, ϕ is the nonlinear activation function and y_k is the output signal of the neuron. This model is sometimes also referred to as a perceptron. The calculation essentially performs a dot product between the input data vector and a weight vector and then adding a bias value before sending the result through a squashing function.

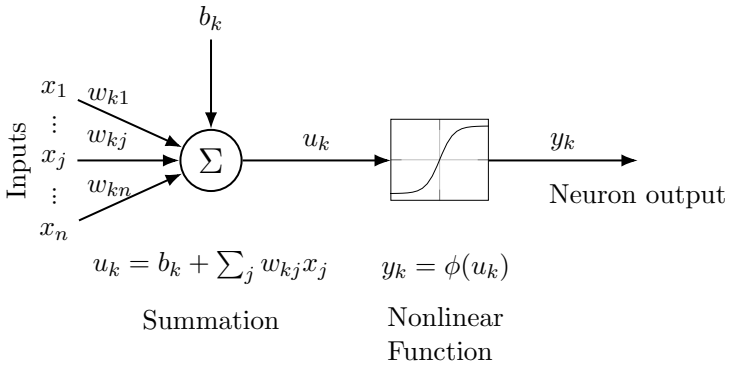

Figure 2.5: Functional schematic of artificial neuron (adapted from [3])

2.4.1 Single layer feedforward network

The artificial neuron acts as a linear classifier that can separate input values into two distinct classes. The classes are separated according to the output value of the activation function (0 or 1). Because the information only flows in one direction from input to output, it is also known as a single layer feedforward network (see Figure 2.6). The network can be trained to separate different input classes by using a simple learning algorithm.

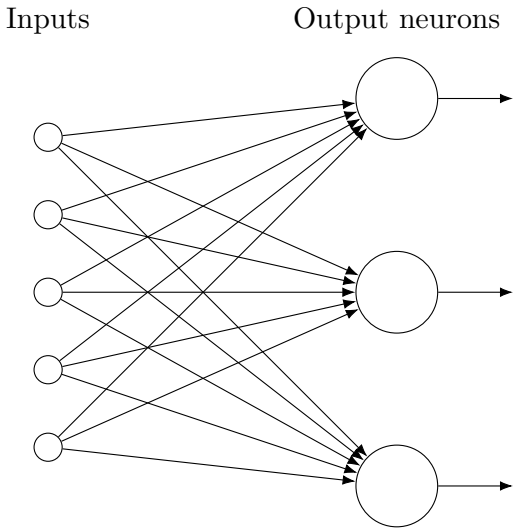

Figure 2.6: A single layer feedforward network.

The problem with this type of network is that it can only separate classes that are linearly separable. When we are given two classes of points in space, A and B, they are linearly separable if they can be separated by a hyperplane, which means there exists a plane where class A and B lie on opposite sides. In Figure 2.7 we provide examples of linearly separable classes in two dimensions in which the hyperplane reduces to a single line. It also shows a configuration of inputs that is not linearly separable and therefore

cannot be classified using a single layer feedforward network.

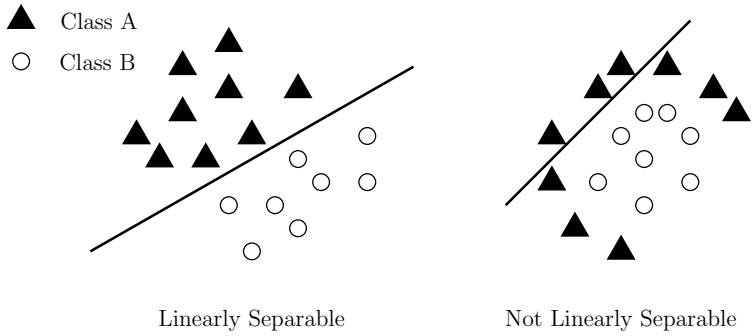

Figure 2.7: Linearly separable and not linearly separable data in two dimensions.

2.4.2 Multilayer feedforward network

We have seen that single layer feedforward networks are limited in the types of data they can classify. To classify classes that are not linearly separable, a different type of network is required, called a multilayer feedforward network. This type of network has one or more hidden layers of neurons between the input and output (see Figure 2.8). In this network the input is sent to the hidden layer and the output of the hidden layer is passed as input to the final layer. The final layer then generates the output of the network.

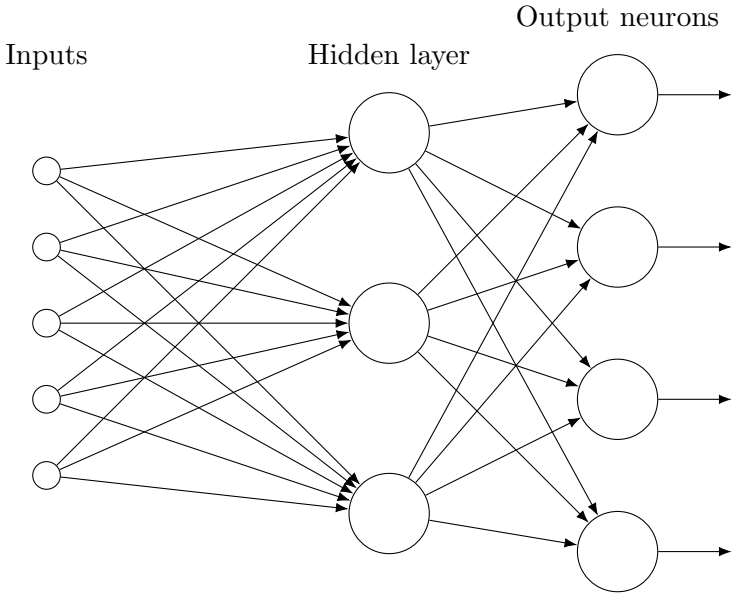

Figure 2.8: A multilayer feedforward network.

An intuitive explanation of what happens can be seen in Figure 2.9. The hidden layer distorts the input space, indicated by the red and blue grid, in such a way that it

becomes linearly separable by the output neuron. The output layer can then classify the data in a similar way to a single layer feedforward network.

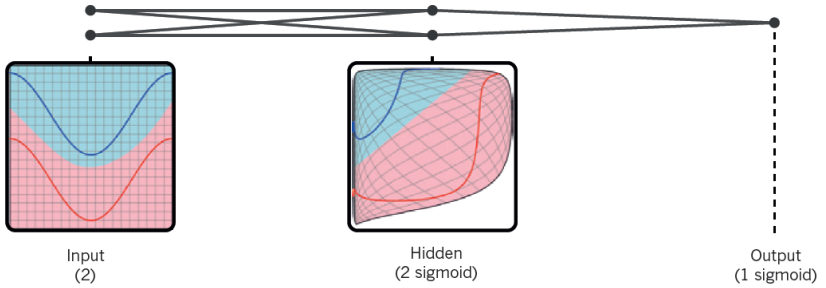

Figure 2.9: Hidden layer distorts input data to make it linearly separable (from [4])

It has been shown [40] that neural networks with a single hidden layer and a nonlinearity such as a sigmoid function can act as universal approximators. This means any bounded continuous function can in theory be approximated with a neural network with a single hidden layer.

In practice, the quality of the approximation depends on the dimensions and number of hidden layers in the network and the way the network is trained. When discussing deep learning we will see there are networks with large numbers of hidden layers that can approximate complex behaviors.

To train the network means we have to determine the value of the weights of the links in order to approximate our function accurately. Because we focus on the inference/classification capabilities of a network, training will only be explained briefly here:

The weights of a neural network have to be chosen in such a way that the network will match the desired output function as accurately as possible over the whole input domain. By using a dataset as input with known outputs, the network can be trained by sending input data through and comparing the network output with the desired output. Using a learning algorithm, the parameters can then be adjusted to achieve the desired results.

A common learning algorithm is backpropagation, which is a method that calculates the error gradient of the network output compared to the desired output and propagates that gradient backwards through the network. In each layer the gradient is calculated with respect to the layer weights by utilizing the chain rule and then propagating the result backwards. Using a gradient descent algorithm, the weights are then continually adjusted to minimize the output error until the desired level of output accuracy is achieved.

2.5 Convolutional Neural Networks

We will now consider a variant of artificial neural networks that is useful for pattern recognition and classification applications. When input data size becomes large, the number of parameters in artificial neural networks increases significantly to the point where it no longer is feasible to create large networks to perform classification. This

is for example the case with images and audio input data. To still be able to take advantage of neural networks with these types of data, Convolutional Neural Networks were created.

Convolutional Neural Networks (sometimes abbreviated to ConvNets or CNNs) are specialized types of artificial neural networks that focus on processing data with a known structure such as a grid. Examples are: audio samples, which can be seen as one-dimensional grids of data and images, perceived as two-dimensional grids of data. In recent years CNNs have been very successful in vision and pattern recognition applications and we will analyze image processing specifically. As with artificial neural networks, convolutional neural networks are inspired by biological processes, specifically the experiments by Hubel and Wiesel on receptive fields in the visual cortex [41].

In a CNN, a successively higher-level abstraction of the input data is generated at each layer. This data is stored in what are known as feature maps that are then fed to the next layer (see Figure 2.10). Deep Neural Networks, sometimes implemented with CNNs, take advantage of the fact that signals in nature are hierarchies of features, where high-level features are composed out of low-level ones. In a CNN the role of the convolutional layer is to extract these features, whereas the pooling layer is used to merge similar features into one output activation which creates invariance against spatial translations and distortions [4, p. 439]. These characteristics are useful for pattern recognition applications where each input can vary in small ways, but still yield the same output.

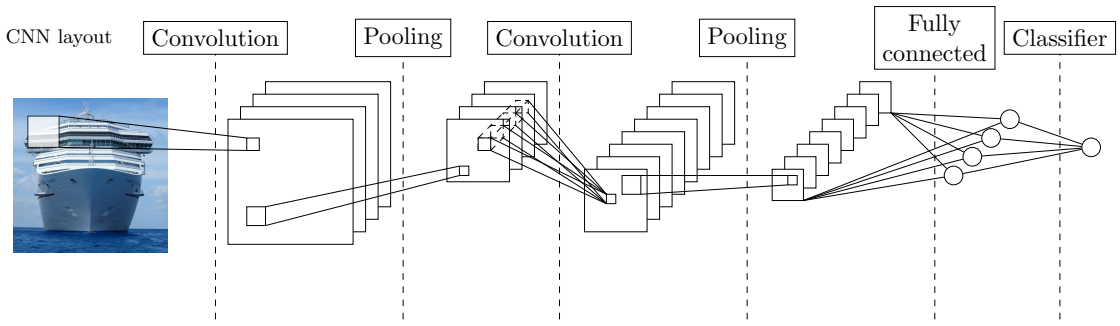

Figure 2.10: Example CNN illustrating typical layer configuration (adapted from [5])

The convolutional layers perform convolution operations on the feature maps. The feature map itself is two-dimensional and convolved with a unique 2-D filter kernel. The result of the convolutions are then summed together into the output feature map and optionally a bias value is added. The operation is illustrated in Figure 2.11 for a single output feature map. Multiple output feature maps can be created by adding more sets of filter kernels.

The intuition behind this is that a filter kernel activates on certain features on the input data. If a kernel applies to a low-level feature such as horizontal stripes, it is useful to apply the same kernel across the entire input, instead of having separate weights for each pixel. This reduces the amount of parameters required for the network significantly.

CNNs can be viewed to be feature extractors. Based on a kernel of weights, a certain feature is extracted from the input data and written to the feature map. In Figure 2.12 some filter kernels of the AlexNet [6] network have been visualized. As can be seen these

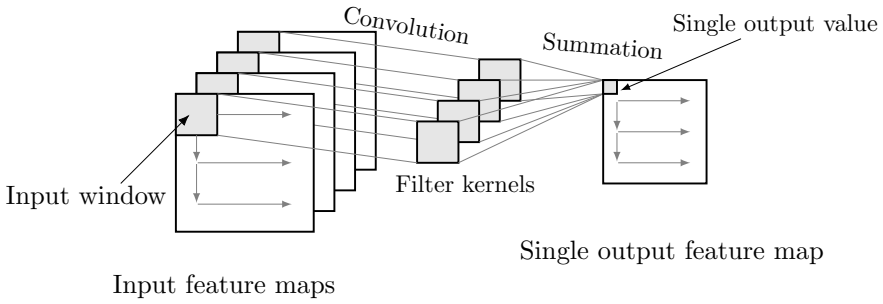

Figure 2.11: Convolution operation of input maps to single output map

are abstract features such as horizontal stripes that will be extracted from the input image into a feature map and sent to subsequent layers.

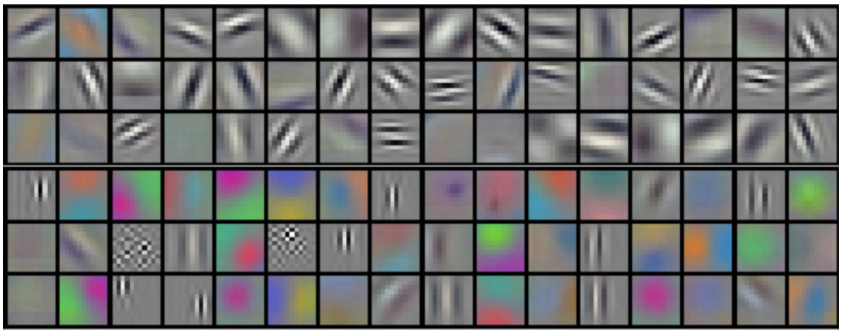

Figure 2.12: AlexNet low-level feature visualization (from [6])

Often, the result of a convolutional layer will be fed through a pooling layer which reduces the amount of data of the feature maps by subsampling the data, this is done by taking a window in the input map and either taking the maximum or average value and sending that to the output map (illustrated in Figure 2.13).

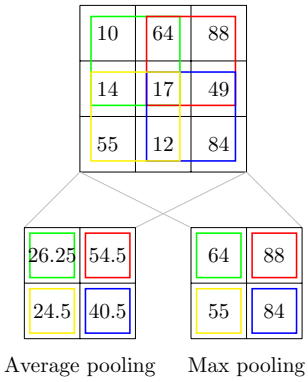

Figure 2.13: Pooling operation (average and max) with kernel 2×2 and stride 1

As in a regular artificial neural network, a nonlinear layer is also added to limit the

amplitude of the output values. In many cases at the end of the network one or more fully-connected layers are added, which are equivalent to the multilayer feedforward network from Section 2.4.2. These are used as the final classification output and because the successive convolution/pooling layers have reduced the input data size, it becomes feasible to pass the data through the fully-connected layers. The CNN layers can be trained in much the same way as the regular multilayer neural network using backpropagation, often utilizing GPUs or dedicated accelerators to accelerate this process.

2.5.1 Common networks

As we have seen, CNNs are a useful tool for pattern recognition applications. Over the years, several successful CNNs have been developed that have seen widespread application both in academia and industry. We will briefly mention several important ones:

- LeNet [8]:
  One of the first successful CNN implementations trained with backpropagation, used for digit classification on 28×28 grayscale images. This network was introduced in 1989 and was used for automated reading of bank checks.

- AlexNet [6]:
  The first CNN to win the ImageNet image classification challenge in 2012. It has five convolutional layers and three fully-connected layers. The number of parameters compared to LeNet is much larger and to reduce training time this network was trained on GPUs. This network achieved a 17.0% top-5 error on the ILSVRC [10] dataset.

- VGG-16 [42]:
  A deep network with 16 layers (13 convolutional and 3 fully-connected) and taking advantage of smaller filters. This network achieved a top-5 error between 8.1% and 8.8% on the ILSVRC [10] dataset.

2.6 Related work

To implement CNNs such as the ones mentioned in Section 2.5.1 in the real world, compromises will have to be made. This could be for example with respect to training time, available hardware resources and required level of accuracy. In this section we will look at related work that applies to the following: how can we implement CNNs in practice, what kind of platforms are suitable and which necessary tradeoffs will have to be made.

2.6.1 Computational workload

We first analyze the workload of common CNNs. In general we assume the highest workload of the CNN is located in the convolutional layers. This assumption can be backed up with results from [5] and [43] showing that the computation of the convolutional

layers takes between 80% and 90% of the total computation time of the network inference. Furthermore, in [44, appx. D] the workload of the LeNet-5, AlexNet and VGG-16 networks is also analyzed and the convolutional layers consume between 82% and 99% of the total computation time of the inference.

Therefore, the focus of accelerating a network should be centered around reducing the computation time of the convolutional layers.

2.6.2 Hardware acceleration

Because most of the time is spent on the convolutional layers, we would like to accelerate the inference on those layers specifically. To accelerate CNN inference, several methods have been developed. We divide them into three categories and explain them briefly.

CPU and GPU acceleration

As we have seen in the previous section, most of the computation time of common CNNs is spent in the convolutional layers. The core operation for these layers is the Multiply-Accumulate (MAC) operation, which can be highly parallelized as the operation depends on input data from the previous layer and independent constant parameters. To accelerate the performance on CPUs and GPUs, several techniques are used such as vector processing with SIMD or parallel thread processing through Single Instruction Multiple Thread (SIMT).

For example, the convolution operation can be mapped to a matrix multiplication operation after which special libraries for hardware optimized matrix multiplication operations can be used to accelerate this. Examples of these libraries are Intel MKL [45] (for Intel CPUs) and cuBLAS [46] (for NVIDIA GPUs). These libraries optimize the mathematical operation for the specific memory hierarchy and available computational units on the hardware platform. Special libraries specifically aimed at accelerating deep learning are also being developed, such as cuDNN [47] and Intel MKL for DNN [48].

FPGA

FPGA-based CNN accelerators have been proposed as an alternative to CPU and GPU acceleration. Because of their energy efficiency and reconfigurability, FPGAs can be an attractive option for this type of application. In [49] an FPGA based coprocessor for CNNs is presented that can be reconfigured based on available hardware resources and achieves a 4× to 8× speedup compared to fixed CPU/GPU/FPGA implementations. In [23] and [24] FPGA CNN frameworks are presented that can perform design space exploration based on available hardware resources and memory bandwidth. In [22] an FPGA-based accelerator is presented that accelerates deep convolutional neural networks with significantly less power usage compared to GPU based accelerators with similar throughput.

Dedicated DNN hardware accelerators

To increase efficiency even further and achieve higher performance per watt, a dedicated accelerator ASIC can be the solution. In [14] memory access overhead is identified as

the main bottleneck for processing CNNs. Because each MAC operation requires three memory reads (parameter, input and partial sum) and one memory write (the partial sum) and each operation has to go through Dynamic Random Access Memory (DRAM) in the worst case scenario, this is the best area to optimize performance with the help of dedicated accelerators.

By introducing a hierarchy of local memories to reduce the movement of data, energy efficiency can be improved. In the past few years dedicated hardware CNN accelerators have been developed to speed up both inference and training stages. Some examples are [50], DianNao [19] and EyeRiss [51]. To compare the characteristics of existing accelerators, [14] presents a taxonomy based on the way data moves within the accelerator. For example, a weight stationary dataflow minimizes energy use from memory reads by keeping a weight parameter in the register file of a processing element as long as possible while data that operates on these weights passes through.

2.6.3 Deep learning software

In Section 2.6.2 we have seen there are several ways to accelerate deep learning applications with hardware. However, to make life easier for developers and researchers, we would like to abstract away some of the low-level details of these implementations. To this end, different software frameworks for deep learning networks have been developed.

Because most networks consist of common building blocks that are part of these frameworks, development time is not wasted on reimplementing these every time which also reduces the risk of software bugs.

Some examples of frameworks are Caffe [52], Torch [53], Theano [54] and Tensorflow [55]. They support programming languages such as C and C++ but also scripting languages such as Lua and Python for network development, as well as high-level network description files. Often, these frameworks also support hardware accelerators to speed up training. We will briefly discuss the Caffe deep learning framework.

Caffe deep learning framework

Caffe [52] is a software framework developed by the Berkeley Vision and Learning Center (BVLC) to aid the development of deep learning applications. It consists of a library of components, models, tools and interfaces that simplify the development of a custom deep learning network based only on a high-level model description. It is also easily extended by means of Python and MATLAB bindings and has support for GPU acceleration, which is useful for research applications.

It has all the required functionality to train CNNs and export their parameters through custom scripts to any data format. Furthermore, it can be used as a verification tool to test other CNN implementations.

2.6.4 CNN on embedded systems

Thus far, we talked about accelerating CNNs on high-end hardware such as desktop CPUs. However, there are also scenarios in which we would like to run a CNN on less powerful hardware such as embedded systems or mobile devices. When implementing

CNNs on such systems, the limits of the hardware begin to play a role in performing computation efficiently. To optimize the efficient computation of CNNs on such constrained systems, several approaches can be adopted. We consider two options:

- *Reduce precision*
  By reducing the computational precision of operations, fewer demands are placed on the hardware. It can lead to higher performance, lower energy consumption and less resource usage. Precision can be reduced with the help of fixed point computation instead of floating point computation, using fewer bits per data type and sharing parameters.

- *Reduce number of operations and network size*
  This includes techniques such as parameter compression, network pruning and other compact network architectures. As this depends more on the specific network architecture, we will not consider this further.

Quantization

Our interest lies in reducing precision and avoiding floating point operations to make it possible to execute CNNs on the ρ-VEX platform. It has been shown [21] that using 16-bit fixed-point values on certain CNN networks results in "little to no degradation" in level of accuracy compared to 32-bit floating point values. This applies both to parameters and data values, which is ideal in our case because the ρ-VEX processor is a 32-bit platform with a 16×16 bit multiplier. This multiplier yields a 32-bit result with limited overhead.

However, the trained network needs to be converted from floating point to fixed point (quantized) after training is complete. To achieve this, we use the Ristretto software extension for the Caffe framework.

Ristretto extension

By default the Caffe framework generates floating point parameters when training a network, but we also want to be able to use fixed point arithmetic with existing networks. The Ristretto [56] extension has been developed for this purpose and is a plugin for the Caffe machine learning framework.

The Ristretto plugin analyzes the numerical resolution of the parameters and output values of the convolutional and fully connected layers of a CNN. It can then replace the floating point values with fixed point precision values. The bit width of the fixed point values can either be specified manually or fine-tuned automatically within a predefined error margin.

2.7 Conclusion

In this chapter we reviewed the required background information for implementing CNNs on the ρ-VEX platform. First, the types of parallelism that can be exploited were

discussed (TLP, ILP, DLP), followed by an explanation of the ρ-VEX platform that can be used to benefit from these types of parallelism.

Machine learning is briefly introduced, after which we discussed artificial neural networks in more detail, especially the rationale behind multilayer feedforward networks and its ability to approximate complex functions.

This gave us enough background information to explain convolutional neural networks and its advantages over a regular multilayer feedforward network due to its successive layers of higher-level abstraction. Next, we analyzed related work by reviewing CNN hardware acceleration in CPU/GPU/FPGA and dedicated accelerators. Finally, we reviewed deep learning frameworks and specific challenges related to running CNNs on performance-constrained hardware.

Concept

3

In Chapter 2 we discussed the background of the ρ-VEX platform and convolutional neural networks. Using this knowledge, we will develop the concept for a CNN inference pipeline in this chapter. It is based around the ρ-VEX architecture and will take advantage of its features to improve performance.

First, in Section 3.1, we will discuss how to map a CNN to a streaming application by reviewing the required components and estimating resource requirements. Subsequently, in Section 3.2, the proof of concept design is explained. Following this, in Section 3.3 we briefly discuss how we will take advantage of parallelism in the CNN to improve performance. Next, in Section 3.4, the design is presented together with the details on how it will operate and we will examine the possibilities of implementing this design on an FPGA platform through hardware synthesis. Finally, in Section 3.5, we present a conclusion and brief summary of the chapter.

3.1 Convolutional Neural Networks as a streaming application

There are several options for implementing CNNs on the ρ-VEX platform. We could create one giant binary to execute the entire network on a single core or create a fixed hardware implementation of the network attached to a ρ-VEX core as an accelerator.

Because we want to use pipelining to improve performance and we also want to maintain flexibility by using softcore processors, we take the approach of mapping the CNN as a streaming application. It means we can create a CNN out of software components that can be assigned to a pipeline consisting of multiple processing elements.

To implement this, we take inspiration from the fpgaConvNet [24] streaming architecture for evaluating CNNs, mentioned in Section 2.6.2. This architecture consists of several building blocks that represent the layers of the CNN and that can be synthesized on an FPGA platform. By varying certain parameters, a design space exploration can be performed of the required performance versus the available FPGA resources.

In this section we will map the CNN building blocks of fpgaConvNet to a software implementation and develop a streaming architecture consisting of ρ-VEX processors that will run these software blocks.

3.1.1 Estimated resource requirements

To develop our streaming architecture, we first would like to get a sense of the resources required. We analyze two different CNNs in terms of computational complexity and memory usage.

For this purpose we analyze networks that perform inference on the MNIST [29] and CIFAR10 [57] datasets. These networks are based on examples in the Caffe distribution, where the MNIST network in turn is based on a simplified LeNet-5 [8] implementation. These are not the most state-of-the-art implementations, but achieve a good level of accuracy with a simplistic design and limited resource requirements. A large part of the memory requirements of these networks are the weight and bias parameters for the convolutional and fully connected layers, therefore we focus on the required number of parameters and buffer sizes.

Network/dataset	**Required parameters**	**Required map buffer (bytes)**	**Total size (buffer and 16-bit fixed point parameters)**
MNIST	430.500	78.776	917 KiB
CIFAR10	145.376	153.674	434 KiB

Table 3.1: Estimated memory requirements of two networks

In Table 3.1 it can be seen that for MNIST and CIFAR10 we require between 434 KiB and 917 KiB of storage. Due to the fact that local memories on FPGA and ASIC are usually limited to hundreds of kilobytes, it would be difficult to store parameters and buffers for MNIST and CIFAR10 in local memory. In that case it would be necessary to investigate a memory bus architecture that could efficiently supply the cores with the required parameters with acceptable performance.

The computational workload estimate was taken from [58] and can be seen in Table 3.2. To obtain an estimate for the performance we can achieve from the ρ-VEX architecture compared to fpgaConvNet, we choose the same clock speed of 125 MHz. In the best case scenario the ρ-VEX can execute eight instructions per cycle, therefore we assume we can perform eight operations per cycle. It results in the throughput values presented in Table 3.2.

Network/dataset	**Workload in GOps for single forward pass**	**Execution time single ρ-VEX core 125 Mhz (8 instr/cyc)**	**Throughput**
MNIST	0,0038 GOps	0,0038 seconds	263,16 samples/s
CIFAR10	0,0247 GOps	0,0247 seconds	40,48 samples/s

Table 3.2: Computational workload estimates for the networks

Improving the latency of the CNN inference can be achieved by taking advantage of the DLP in the convolution operation. Because all convolution operations inside a convolutional layer are independent of each other, it would be desirable to execute them all in parallel.

As the convolution consists mainly of multiply-accumulate operations, we want to also execute as many as possible of these operations in parallel. However, we have to take into account the constraints on the memory bandwidth and available computational

resources. In our case, we can perform operations on four contexts in parallel on a single ρ-VEX core and use either a limited amount of local memory or external memory available through a bus.

3.1.2 Standard layers

Now that we have a better view of the memory and computational requirements, we need to understand which components will be required to implement the CNN.

We identify several layers that commonly occur in CNN type networks. Based on this subset we will create several software building blocks that can be concatenated to form a CNN. In this section we will discuss these layers, but other types exist and are often tailored to a particular network design.

Convolutional layer

In the convolutional layer the input is usually data stored in a grid form in one or more feature maps. A convolutional layer takes as its input the feature maps containing this data and subsequently outputs one or more feature maps.

Each output feature map is calculated by convolving the input feature maps with kernels of weight values. Each output/input map combination uses one weight kernel for the whole map, in contrast to a regular neural network that would use a different weight for each input/output pixel pair. This reduces the memory requirements compared to regular fully connected layers. Pseudocode describing the operation performed by the convolutional layer is presented in Listing 3.1.

```
1  for (r=0; r<R; r++){            // output feature map
2    for (q=0; q<Q; q++){          // input feature map
3      for (m=0; m<M; m++){        // output x in map
4        for (n=0; n<N; n++){      // output y in map
5          if (q==0) { Y[r][m][n] = Bias[r]; }
6          for(k=0; k<Kw; k++){            // kernel x loop
7            for(l=0; l<Kh; l++){          // kernel y loop
8              Y[r][m][n] += W[r][q][k][l] * X[q][m+k][n+l];
9            }
10         }
11       }
12     }
13   }
14 }
```

Listing 3.1: Pseudocode for convolutional layer operation (from [59])

For square feature maps and kernels the time complexity becomes $O(R \cdot Q \cdot M^2 \cdot K^2)$, where R is the number of output maps, Q the number of input maps, M the output map dimension and K the kernel dimension.

Pooling layer

The pooling layer is used to subsample the feature map and reduce the sensitivity of the output to shifts and distortions [8]. The most common types of operations in pooling layers are max-pooling and average-pooling. In max-pooling we take the maximum of a window of the input data and send this value to the output. In average-pooling the average of an input window is calculated and sent to the output. The pseudocode for max-pooling is presented in Listing 3.2.

```
1  for (o=0; o<No; o++) {                  // output feature map
2    for (m=0; m<Nm; m++) {                // row in feature map
3      for (n=0; n<Nn; n++) {              // column in feature map
4        for (k=0; k<Nk; k++) {            // row in pooling kernel
5          for (l=0; l<Nl; l++) {          // column in pooling kernel
6            accumulate = MAX(accumulate, input[o][(Sm*m)+k][(Sn*n)+l
                 ]);
7          }
8        }
9        output[o][m][n] = accumulate;
10     }
11   }
12 }
```

Listing 3.2: Pseudocode for MAX pooling operation

For square feature maps and kernels the time complexity becomes $O(Q \cdot M^2 \cdot K^2)$, where Q is the number of output maps, M the output map dimension and K the kernel dimension. The number of output and input maps is always equal for this layer.

Nonlinear layer

The nonlinear layer turns the feature maps into activations by passing the input data through a nonlinear function. It is similar to the nonlinear activation function in regular artificial neural networks we have seen in Section 2.4.

The inputs are squashed between 0 and 1 (or -1 and 1) and are sent to the output. The functions used in artificial neural networks are used in CNNs as well, such as the logistic function, hyperbolic tangent (TanH) and Rectified Linear Unit (ReLU). Currently ReLU is one of the most often used functions because of its simple computation and good convergence characteristics during network training. The different functions are graphed in Figure 3.1.

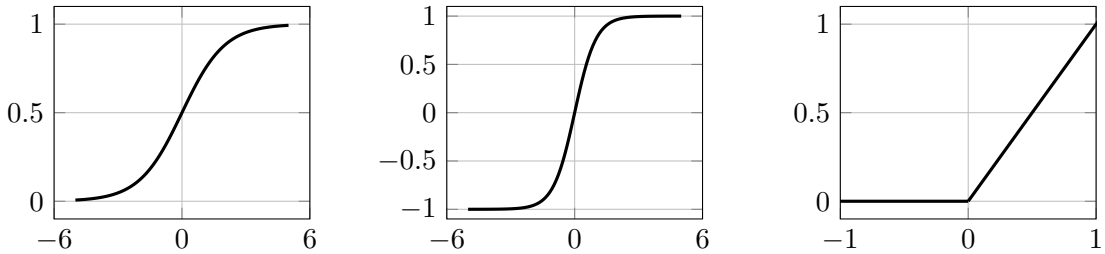

Figure 3.1: Common activation functions

```
1 for (o=0; o<Nf; o++) {                // output feature map
2   for (m=0; m<Nm; m++) {              // row in feature map
3     for (n=0; n<Nn; n++) {            // column in feature map
4       output[o][m][n] = MAX(0, input[o][m][n]);
5     }
6   }
7 }
```

Listing 3.3: ReLU operation pseudocode

In Listing 3.3 the pseudo-code of the ReLU layer is presented, note that the ReLU function is implemented as $MAX(0, input)$. The time complexity of this is $O(Q \cdot M^2)$ where Q is the number of output maps and M the output map dimension. For this layer the number of output and input maps is always equal.

Fully connected layer

The fully connected layer is used as a final classifier at the end of the network. Because of the required amount of parameters, it is only feasible when the size of the input data of the network has been significantly reduced, which was achieved through successive convolutional and pooling layers in front of this layer.

This layer is equivalent to the regular neural network layer seen in Section 2.4. It can also be viewed as a fully connected convolutional layer with 1×1 kernel sizes and 1×1 input/output feature map dimensions. Pseudocode describing the operation of the fully connected layer is presented in Listing 3.4.

```
1 for (r=0; r<R; r++){              // outputs
2   for (q=0; q<Q; q++){            // inputs
3     if (q==0) { Y[r] = Bias[r]; }
4     Y[r] += W[r][q] * X[q];
5   }
6 }
```

Listing 3.4: Fully connected layer pseudocode

The time complexity is $O(R \cdot Q)$ where R is the number of output neurons and Q the

number of input values.

3.1.3 Implementing CNN building blocks as software components

Based on the representations in the previous section, we can now determine which blocks to implement in software. Because the CNNs we would like to implement are feedforward neural networks and we do not consider back propagation, the network can be modeled as a Directed Acyclic Graph (DAG).

In [24] the CNN is represented as a Synchronous Dataflow [28] model to aid the analytical modeling. It simplifies the CNN to a DAG with the nodes representing the computation and the arcs the dataflow. In this way, the CNN inference is reduced to a streaming problem.

In fpgaConvNet, the user specifies a high-level description of the network together with a description of the targeted FPGA platform. It then creates a DAG of the network and produces a hardware representation based on the given platform constraints. Our goal is to provide the same CNN building blocks in such a way that a network description for fpgaConvNet can also be implemented as a ρ-VEX stream. Because of the different nature of the implementation, required hardware constraints will not be taken into account.

To achieve this goal, the building blocks will have to be developed as software functions that can be run on the ρ-VEX processor. These can then be combined by means of a code generator that takes in a high-level description of the network and produces the C code that will evaluate the network.

A brief overview of the implemented blocks is presented here:

Sliding window block

This is a sliding window that moves over the input data and outputs a window of the data at each loop iteration. We feed this into other blocks for processing such as the convolution that can then directly convolve a kernel with the window.

Convolution bank

This function takes in a window of input data and convolves it with a kernel based on the current input map and stores the result in an accumulated value in the output stream.

Pooling bank

This function takes a window of input data and performs a pooling operation, currently only MAX pooling and AVERAGE pooling are supported. The result is sent to the output stream.

Nonlinear bank

This function takes an input stream and applies a nonlinear function to each input value and write it to the output stream. Supported functions are ReLU, Sigmoid and TanH.

Other blocks such as a fork unit and memory unit from the fpgaConvNet paper will not be implemented as they are not relevant to our software implementation.

3.2 Single core proof-of-concept

Before we implement the aforementioned building blocks, we would like to examine the feasibility of running the CNNs on the ρ-VEX platform. To this end we first develop a proof of concept by creating an application that can run an MNIST network on a single core 8-issue ρ-VEX. It will be tested on both the simulator and as a hardware implementation on an FPGA development board.

The standard ρ-VEX simulator described in Chapter 2 can be used to run the application. By utilizing the framebuffer plugin, the input and output of the network can also be visualized.

For the hardware implementation we use the existing ρ-VEX platform that is connected to the external DRAM of the ML605 development board and has local instruction- and data-caches. By using the external DRAM, there are no limitations with regard to storing the parameters and buffers for the network that is used. The network input and output can also be visualized with the aid of a Video Graphics Array (VGA) framebuffer on the board.

3.3 Taking advantage of CNN parallelism with ρ-VEX

The reason for implementing CNNs on the ρ-VEX is that we can use its features to improve performance by exploiting parallelism. We will briefly discuss how we can take advantage of the parallelism in the CNN with the streaming pipeline.

Reconfiguring the cores based on component TLP/ILP

We can adapt each core based on the ILP and TLP that is available in the implementation of that specific layer. By measuring the ILP in advance, the optimal issue-width and number of contexts can be determined for a core. Based on this, the number of contexts for each core should be configurable in a simple way without having to rewrite the code.

Increasing performance with parallel streams

To increase the throughput in a simple manner would be to have multiple parallel streams operating simultaneously. By operating on multiple data at the same time the throughput can be increased in a straightforward way.

Combining layers

As we have seen in Section 2.6.1, computation times vary across layers, where convolutional layers take up around 80%-90% of total computation time of the CNN. If we map each operation to a separate core, this would create cores that are idle a lot of the time because of the significant difference in computation times between operations.

To prevent idle cores and reduce wasted resources, some layers could be grouped together on a core without impacting the throughput significantly. In Chapter 5 we will measure the influence on performance and resource usage by grouping these layers together.

Loop unrolling

Our convolution implementation is implemented as four nested loops, one loop over the input feature maps, one over the output feature maps and two loops over the convolution kernel dimensions. It has been shown [30] [60] that loop unrolling can improve performance because it reduces the branch penalties and pointer arithmetic with respect to the loop index. It also improves data independence of instructions, which can increase the possible ILP. This is beneficial for wide VLIW architectures such as the ρ-VEX.

3.4 Creating a ρ-VEX streaming pipeline

We have presented the individual network components that need to be implemented and the methods for exploiting parallelism, giving us all the components we need to continue our design.

In this section we develop the streaming pipeline for executing the CNN inference. First, we discuss the motivation behind the design after which we present it in detail. The general design aspects are reviewed such as synchronization and storage of parameters. This is followed by an explanation of specific details relating to the simulator and hardware implementation.

3.4.1 Motivation

In the subset of CNNs that we consider the layers depend only on the previous layer for input data, this structure can be exploited with pipelining to improve the throughput of the network. This is achieved by placing multiple ρ-VEX cores in series and running each layer of the network on its own dedicated core and streaming the data through the cores horizontally. By placing double buffers between each layer, the cores can operate on the data in parallel through pipelining. The mapping between CNN and cores is illustrated in Figure 3.2.

To improve the latency of each layer, the processor can adapt to the available ILP and DLP in the code of the layer implementation. By changing the issue-width of the ρ-VEX core, it can be optimized for the current code that is running. If there is not enough ILP available we can take advantage of the data level parallelism and divide the core up into at most four contexts that each can process a chunk of data in parallel.

3.4.2 General design

We now present the general design of the streaming pipeline that will be implemented both in the simulator and as a hardware implementation.

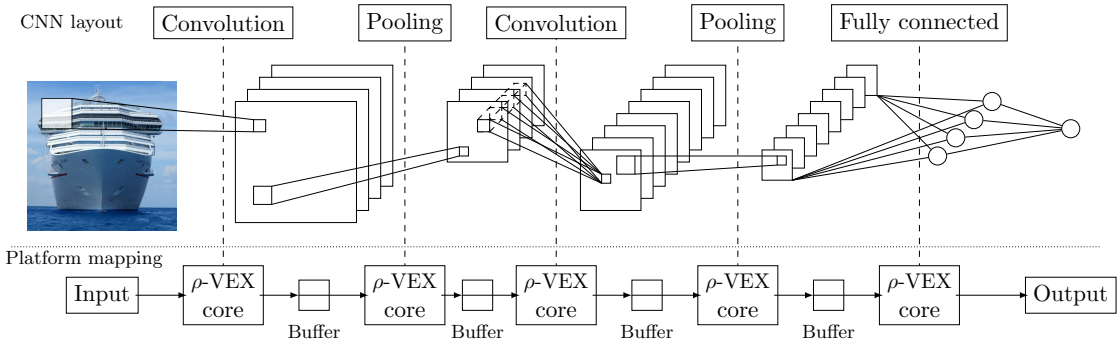

Figure 3.2: Layout concept (partially adapted from [5])

Stream design

To connect the cores together in a stream, we place buffer memories between each core pair that can accommodate the entire output of a layer multiplied with the amount of buffering that we want to do for pipelining. In practice this means the buffer can accommodate two full data outputs in such a way that the previous layer can store the new data, while the next layer operates on the old data.

The memory is operated as a First-In-First-Out (FIFO) buffer in such a way that the new data is always positioned after the last written data and then wrapped around when the end of the buffer is reached. An example of this configuration is presented in Figure 3.3 using three cores with a different number of contexts each.

To implement the FIFO functionality, the core needs to be aware of where it should read and write the data. For this purpose each core has two FIFO registers, a WRITE pointer register and a READ pointer register. The WRITE pointer points to the last written data and the READ pointer to the last read data. When the READ and WRITE pointers are on the same address, the FIFO buffer is either full or empty. To differentiate these cases, the Most Significant Bit (MSB) of the register is flipped every time the address wraps around. Now the buffer is empty when the addresses and their most significant bits are equal and the buffer is full when the addresses are equal but the MSB is different. The FIFO functionality is illustrated in Figure 3.4.

The FIFO read/write registers are located in memory as to not further complicate the ρ-VEX design and are indicated in Figure 3.3. In this way the cores can synchronize their data and operate in a pipelined configuration.

Parameter storage

To reduce the latency of memory to a minimum, all parameters are stored locally in the cores. The amount of storage required per core depends on which layer will be executed on the core. Based on the layer arguments, these memories can be accurately sized. They can be loaded with the appropriate values at start time and do not change after that.

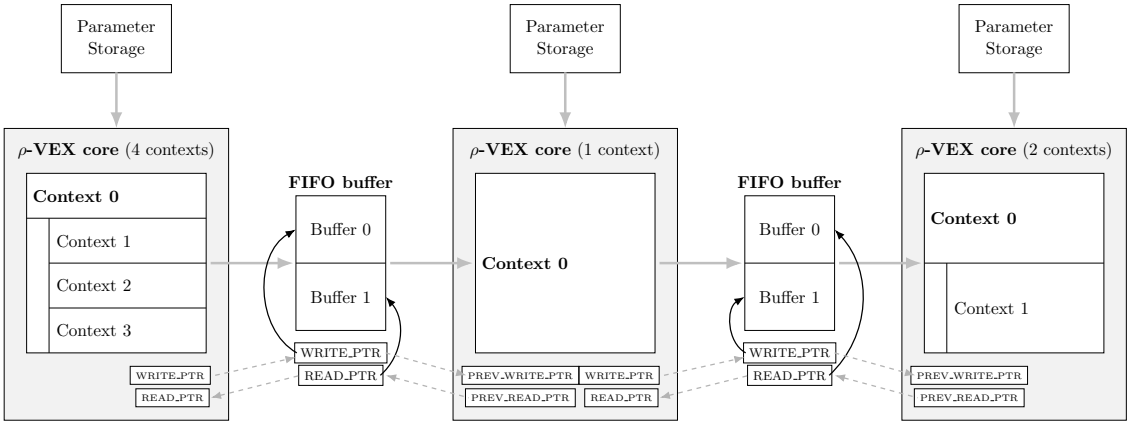

Figure 3.3: Example of a three core configuration

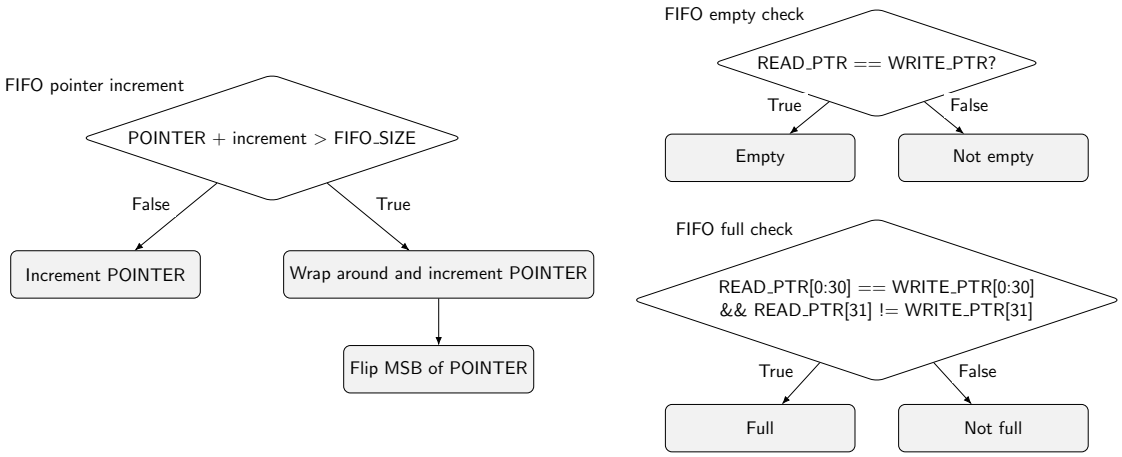

Figure 3.4: FIFO register operation and states

Synchronization

The FIFO buffers are used to synchronize the transfer of data between the cores, we now explain this in more detail. When the FIFO buffer is empty or full, the core has to wait until new data is available or processed, respectively. Currently, a BUSY-WAIT loop is used for this purpose, which constantly checks the FIFO availability.

When using multiple contexts, the FIFO WRITE pointer can only be incremented when all contexts complete their operation on the data. Because the ρ-VEX currently lacks a load-link/store-conditional type instruction that is required to implement atomic primitives such as semaphores for thread synchronization, a simple BUSY-WAIT loop is used here as well.

In a multiple context scenario, context 0 takes care of incrementing the FIFO registers and signaling the other contexts. This is illustrated in Figure 3.5. When data is ready (FIFO is not empty), context 0 sets start flags for the other contexts to true and reconfigures the processor to enable the desired number of contexts. These contexts start executing on their subset of data and then set a completion flag. Context 0 per-

forms a BUSY-WAIT loop on the completion flags and when they are all set to true it disables the other contexts again and writes the output data to the FIFO buffer. There is the possibility that hazards exist in the BUSY-WAIT loop due to the lack of atomic operations, but for our purposes this implementation can still be used.

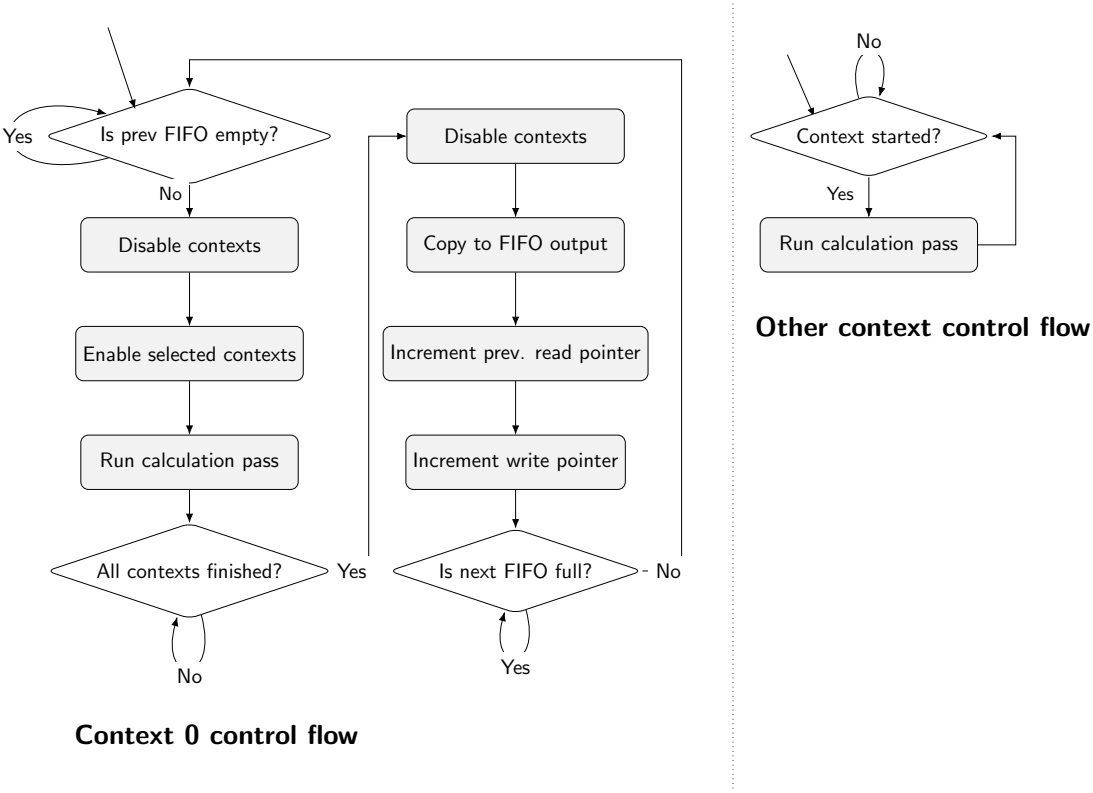

Figure 3.5: Context control flows

For a more energy efficient implementation, it would be beneficial to implement an interrupt system in the FIFO buffer that triggers when the read and write registers are changed. In this way a wake-on-interrupt routine could be implemented in context 0 that would restart the core when FIFO data is available again. Currently context 0 is always running even when waiting for data and this could be considered inefficient. As we do not consider power consumption as a performance metric, the interrupt is not part of the current design.

Resource requirements for MNIST network

To implement the MNIST network on the streaming pipeline, an adequate mapping has to be found between cores and CNN layers. A four core configuration was chosen to minimize the number of resources, while maximizing the pipelining performance. The reasoning being that the most computationally intensive operations, the convolutions and fully-connected layers, are each executed on a separate core, thus taking advantage of the pipelining.

The other operations such as pooling and ReLU are combined with the convolution operations as they do not contribute significantly to the computation time. If these would have also been executed on separate cores the amount of resources would increase significantly while these cores would be mostly idle due to their short run time compared to the convolution operations. This is further explored in Section 5.5.3.

By considering the requirements on parameter size, code/binary size and data buffers, the local memories can be appropriately sized. Due to the design of the ρ-VEX, the memories are allocated in increments of powers of 2. Assuming 16-bit fixed point datatypes, the following sizes have been determined for the instruction memory in Table 3.3. Similarly, based on the required buffers for storing data, the sizes of the data memory for each core have been determined in Table 3.4.

	Operation	**Parameter size**	**Binary size**	**Total**	**Allocated size**
Core 0	Conv1, Pool1	1.000 bytes	48 KiB	50.152 bytes	64 KiB
Core 1	Conv2, Pool2	50.000 bytes	48 KiB	99.152 bytes	128 KiB
Core 2	FC1, ReLU	800.000 bytes	48 KiB	849.152 bytes	1024 KiB
Core 3	FC2	10.000 bytes	48 KiB	59.152 bytes	64 KiB
			Total required		**1280 KiB**

Table 3.3: Instruction memory sizes

	Operation	**Buffer size**	**Allocated size**
Core 0	Conv1, Pool1	1.568 + 23.040 + 5.760 = 30.368 bytes	32 KiB
Core 1	Conv2, Pool2	6.400+1.600 = 8.000 bytes	8 KiB
Core 2	FC1, ReLU	1.000+1.000 = 2.000 bytes	2 KiB
Core 3	FC2	20 bytes	1 KiB
		Total required	**43 KiB**

Table 3.4: Data memory sizes

We have presented the general design of the streaming pipeline and will now review the specific design details required for the simulator and hardware implementation.

3.4.3 Simulator specific details

Initially, we will implement the pipeline in the simulator to test viability and create a simple way to debug the implementation. Because the amount of parameters in the

convolution kernels of a CNN can be significant, an efficient memory architecture is required to ensure the memory does not become a bottleneck in supplying data to the ρ-VEX core.

In the simulator we will divide a shared memory block up into smaller blocks that act as local memories for the cores. Each core is only allowed to access its own local block and the block of the previous core. The previous core can be accessed by setting the MSB of the address to 1. This is illustrated in Figure 3.6.

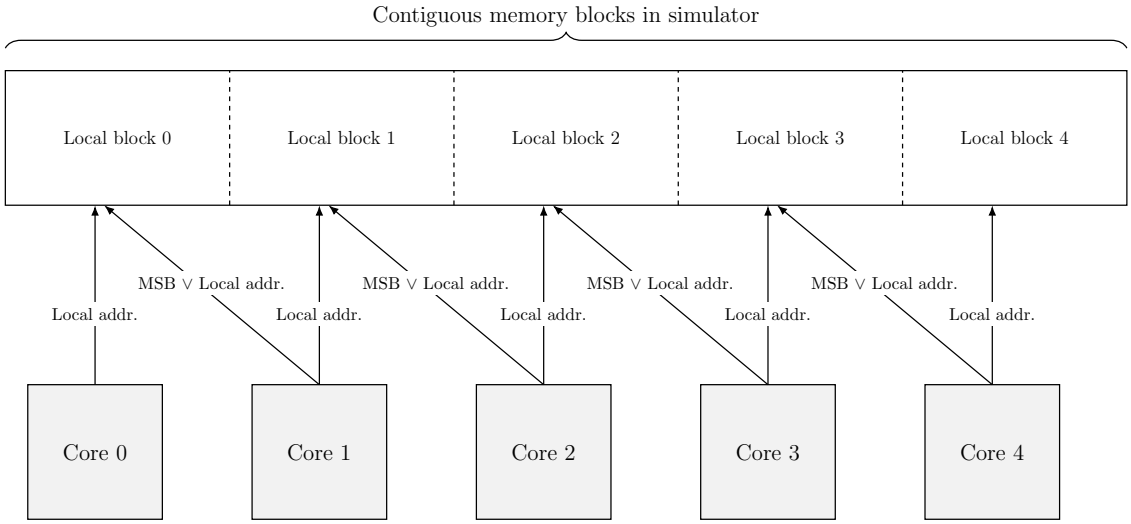

Figure 3.6: Simulator memory layout with previous core access through setting MSB

The FIFO buffers are implemented in such a way that they can be controlled by the software and are independent of any underlying architecture. The FIFO registers are implemented as memory locations in the simulator and are accessed by means of load/store instructions. We will now discuss hardware specific details of the design.

3.4.4 Hardware specific details

Aside from the simulator, we also would like to implement the streaming pipeline in hardware by mapping it to the streaming fabric. To achieve this, we need to take into account certain characteristics of the underlying hardware architecture that we will discuss in this section.

Streaming pipeline

The ρ-VEX streaming fabric architecture mentioned in Section 2.2.3 can be used to demonstrate the streaming implementation. These consist of simplified 2-issue width ρ-VEX cores, so adapting to ILP/TLP cannot be tested because the core cannot run more than one 2-issue context. By synthesizing appropriately sized local memories for each core, a network could be executed on a stream. In Figure 3.7 this is illustrated together with the memory requirements for the instruction- and data-memories.

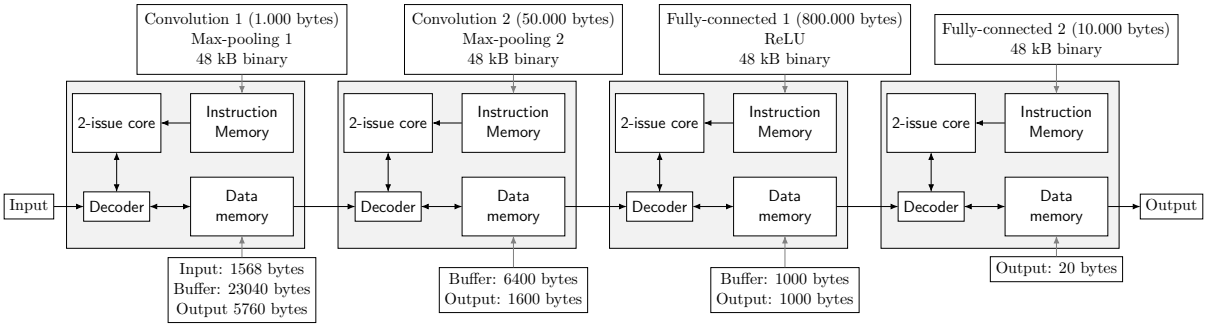

Figure 3.7: Hardware implementation of the network with streaming cores (adapted from [7])

Larger networks cannot fit into the local memories of the FPGA and would require an external memory interface to stream the parameters through the cores at the correct time. However, this could lead to bottlenecks in the memory access and increase design complexity. For now we will therefore only use local memories with smaller networks.

To implement appropriately sized local memories, changes will have to be made to the VHDL files of the streaming fabric. Fortunately, the ρ-VEX cores are designed in such a way that by changing only two parameters, the sizes of the data memory and instruction memory can be configured. Changes will need to be made to the debug bus as well to be able to access all memory, as the addres decoding needs to be mapped differently due to the changed sizes of the local memories in each core.

Implementing FIFO registers

In order to implement the FIFO registers in a hardware implementation, we would have to only modify the memory interface and the processor does not have to manage extra registers in the instruction set. The values can be stored in a BRAM or Look-up Table (LUT) block inside the memory interface. The memory interface can then filter out the addresses to read or write these registers via load/store instructions.

To satisfy the constraint that BRAM/LUT blocks only have one write port and two read ports, the FIFO registers and buffer data have to be read-only for one of the cores. To solve this, the READ register of the current core is read-only for itself and read-write for the next core and the WRITE register is read-write for the current core and read-only for the next core. This satisfies the “two read ports, one write port” constraint and still maintains the required functionality.

3.5 Conclusion

In this chapter we have presented a design of a pipeline of 8-issue ρ-VEX cores for CNN inference. We discussed the individual CNN components that will have to be implemented as well as their representation in software. We converted the fpgaConvNet building blocks to software implementations in order to map the CNN to a streaming representation. To prove the viability, we proposed a single core proof of concept

implementation of CNNs on the ρ-VEX, both in simulation as well as on an FPGA development board.

Next, we briefly discussed some options for taking advantage of the parallelism of CNNs using this pipeline, such as running multiple streams in parallel, combining multiple layers on a single core and loop unrolling.

Multiple streams are a simple way to improve the throughput of the network by just cloning the stream. Some layers can be combined without affecting the throughput significantly because they have lower computation time. Loop unrolling can be used to improve the ILP of the code by reducing loop index pointer arithmetic and branch penalties, this can be especially useful for the convolutional layer, as it consists of four nested loops. Several optimizations are further explored in Chapter 5.

Subsequently, we presented the design of the streaming pipeline. We offered an explanation of how the design will operate with regard to memory and synchronization using the FIFO buffers. By mapping the individual CNN layers to ρ-VEX cores with FIFO buffers in between, a pipeline can be created to improve the throughput. By employing a simple BUSY-WAIT loop we worked around the lack of atomic synchronization primitives in the ρ-VEX.

We analyzed the different details related to the simulator implementation, which mainly revolve around the memory layout and FIFO register. Finally, the possibilities with regard to hardware synthesis are examined by using the single core ρ-VEX design and the ρ-VEX streaming fabric.

4 Implementation

In the previous chapter we discussed the design of our streaming architecture for CNNs. We reviewed the components required and what general design choices were made as well as specific details relating to the simulator and hardware design.

We continue in this chapter with the implementation of the streaming architecture for CNNs on the ρ-VEX platform. First, in Section 4.1, we discuss a proof of concept implementation that runs a simple CNN on the single core ρ-VEX simulator and FPGA platforms. Subsequently, in Section 4.2, we show how the pipeline is implemented and what steps have been taken to achieve this. Following this in Section 4.3, we go into more detail about the operation of the code generator and how the high-level description is mapped to code. Next, in Section 4.4, the efforts with regard to hardware synthesis are reviewed and finally in Section 4.5 a conclusion and brief summary are given.

4.1 Proof of concept

To determine the feasibility of CNNs on the ρ-VEX, a proof-of-concept CNN implementation of a network that is trained with the MNIST dataset [29] was developed.

Using the reference MNIST example network from Caffe, the network was quantized to 12-bit (8.4) fixed point with the Ristretto extension. It runs both on the simulator and on the FPGA implementation of the single 8-issue ρ-VEX core.

The implementation was tested on the ML605 [61] FPGA board using the existing ρ-VEX platform that connects to the off-chip DRAM on the board. Both the simulator and the FPGA implementation achieved the same level of accuracy on the entire MNIST test set as our Caffe reference implementation (99.24%). To achieve this, it was required that the output of the ρ-VEX implementation was equal to the reference output from Caffe/Ristretto.

The biggest difficulty here lies in the fact that due to the way Ristretto operates, a rounding operation needs to be performed after each computation. To get the exact same data out of the ρ-VEX implementation, we need to implement this rounding step as well. Because Ristretto uses the built-in `round()` function for floating point datatypes from the standard C/C++ library, we need to implement the same function in 16-bit fixed-point on the ρ-VEX. After experimentation with several functions, the final rounding implementation is presented in Listing 4.1.

```
1  int16_t round(int16_t a) {
2    if (a == 0) {
3      return a;
4    } else if (a > 0) {
5      return (a + (a & RND_ADD)) & RND_AND;
6    } else {
7      return -((-a + (-a & RND_ADD)) & RND_AND);
8    }
9  }
```

Listing 4.1: Implementation of the rounding function

In this function arithmetic rounding away from zero is implemented. By masking the first fractional bit, the value 0.5 will be added to the original value based on the condition that the fractional part is equal or larger than 0.5. After the addition the fractional part is masked off and the rounded value returned. The same applies to the negative values, by first converting it to a positive number the rounding away from zero can be implemented, however this is not efficient due to the amount of sign conversions taking place and could be improved.

A demonstration application can be seen in Figure 4.1 that runs the same CNN code. It visualizes the input data as well as the inferred output value, indicating correct values in green and errors in red. This application runs both on the simulator with the aid of the framebuffer plugin, and on the FPGA implementation by using the VGA interface framebuffer. In this way a working prototype has been demonstrated, giving a clear picture of the feasibility.

Figure 4.1: Demonstration of MNIST network (simulator)

4.2 ρ-VEX streaming pipeline in simulator

After implementing the proof-of-concept, we will discuss the implementation in the simulator. To implement the streaming pipeline in the simulator, the most important change that had to be made was to add support for multiple cores.

The original simulator is designed for single core processing and consequently the processor state was distributed throughout the source code. To implement a multi-core simulator, all the processor state and caches are combined in a single C language structure that can be easily referenced by means of memory pointers. In this way most functions can remain the same and the only changes that need to be made are that the functions need to accept a pointer and any core state references are accessed through that pointer.

When the simulator starts, each core is assigned a structure containing the entire state. The binary for each core is loaded into the core memory and the state is initialized. The cores are then started and the main simulator loop begins. The simulation loop executes a single cycle of each core in round-robin fashion to ensure that memory stays coherent. In each cycle, the core also executes its own internal configured contexts in a round-robin manner. When all cores complete their execution, the loop terminates and the simulator dumps the final state of each core to the terminal.

This simulator design limits the performance in cases where a large number of cores are simulated, because a single simulator thread is running all cores. An effort was made to improve the performance of the simulator by running each core on a different thread using Pthreads [62] to take advantage of multi-core host systems.

Because we require synchronization of the threads after each simulated cycle, a synchronization barrier was added. It created such an amount of overhead in the performance that no gains were achieved and the multithreading effort was abandoned. To improve the performance, a more detailed analysis would have to be made on how coherency can be maintained while simulating multiple cycles in parallel.

The original simulator is configured via command line arguments which is beneficial when automating simulations with the aid of command line scripting. However, for multiple cores the number of command line arguments increases and becomes unwieldy, therefore a configuration file loader was created that loads a configuration file with all the parameters for each core. For example, the binary and Random Access Memory (RAM) images required by a core can be specified in this file. An example three core configuration is provided in Listing 4.2.

```
1  core=0
2  binary=core0.elf
3  ram=0x1E00000,data_images.ram
4  ram=0x3002028,params/layer_w0
5  ram=0x3002000,params/layer_b0
6
7  core=1
8  binary=core1.elf
9
10 core=2
11 binary=core2.elf
12 ram=0x3002474,params/layer_w1
13 ram=0x3002410,params/layer_b1
```

Listing 4.2: Core configuration file with binaries and ram images

As a memory model, each core gets assigned a block of memory from a larger memory pool that is allocated at the start time of the simulator. The binary of each core gets loaded into this block and is also used as general purpose memory for the core, but care must be taken that the binary is not overwritten by the code by choosing appropriate memory offsets and sizes. The memory model is illustrated in Figure 3.6.

The memory interface of the simulator was also adapted to incorporate the FIFO buffers and registers. When a core accesses a memory address with the MSB set, the access is directed to the memory of the previous core in accordance with the design of the ρ-VEX streaming fabric. In this way the core can address both memories using the same address ranges. The FIFO registers themselves are placed at fixed high addresses to minimize possible conflicts. As explained in Section 3.4.4 the READ register is read-only for the current core and the WRITE register is read-only for the next core. To enforce this behavior, the simulator will generate a memory fault when a core tries to write to a read-only FIFO register.

Now that the implementation of the pipeline in the simulator is complete, we can turn our attention to implementing the software that runs on it.

4.3 CNN code generation

In this section we discuss the development of the code generation. First we briefly review how we chose the programming languages for the implementation. After that we present the different software components that are required for the code generator and finally how the implementation operates in practice.

4.3.1 Programming languages

Python scripting language

The Python [63] scripting language was chosen for creating the code generator. Because of its simple syntax and powerful third-party libraries, this was the fastest method of creating a generator for our code. With the PyParsing library [64] we can easily create

a parser for the high-level description of our CNN network. All the C code files are then generated from Python, as well as the Makefile and core configuration file.

C programming language

The only language that is currently well-supported by the ρ-VEX platform is the C programming language [65]. This language is commonly used for embedded systems and low-level programming because of its flexibility when it comes to manipulating memory and its high performance. The drawback is that mistakes are easily made when manipulating memory directly and the learning curve of this language can be quite steep.

4.3.2 Building blocks

The CNN layers are implemented as software building blocks. Each block consists of a function that has an input and output stream pointer argument, as well as arguments for the various parameters relating to the specific function. For example, the convolutional layer has parameters for feature map dimensions, kernel dimensions and pointers to the weight and bias values. Each block also has an argument for the current timestamp of the stream, this timestamp increases for each loop iteration, indicating which data point needs to be operated on next.

The following blocks are implemented:

- *Sliding window*
  This block can take a window of data and slide it across the input stream, the window of data is sent to the output stream. The current window being output depends on the timestamp argument.
  Arguments: input stream, output stream, kernel dimensions, stride, input dimensions, zero padding, number of input maps, timestamp.

- *Convolutional bank*
  This block performs the operation of the convolutional layer and can be divided into multiple banks across contexts to operate in parallel. The workload is spread by giving an extra offset argument to indicate which output maps a context has to operate on. The input to this layer originates from a sliding window block.
  Arguments: input stream, output stream, number of input maps, number of output maps, output map offset start, output map offset end, output dimensions, kernel dimensions, pointer to weights and bias parameters, timestamp.

- *Pooling bank*
  This block performs the pooling operation, max-pooling and average-pooling were implemented. The input to this layer originates from a sliding window block.
  Arguments: input stream, output stream, kernel dimensions, input dimensions, timestamp.

- *Nonlinear-bank*
  This block performs the nonlinear operation in the network. Each value on the

input stream is sent through the nonlinear function and subsequently to the output stream. The ReLU, TanH and sigmoid functions are implemented, TanH and sigmoid by means of a lookup table.
Arguments: input stream, output stream, timestamp.

Because each function has an input and output stream, they can be chained to implement a CNN. Either they are all connected within the same core, or spread over multiple cores through the FIFO buffers. We next examine how the code generator uses these blocks to generate all code necessary to run the network.

4.3.3 Component architecture

The code generator consists of a parser (`generator.py`) that loads the high-level description and two files (`core_gen.py`, `block_gen.py`) that generate the code for the CNN operations and the main code for the ρ-VEX core, this is illustrated in Figure 4.2.

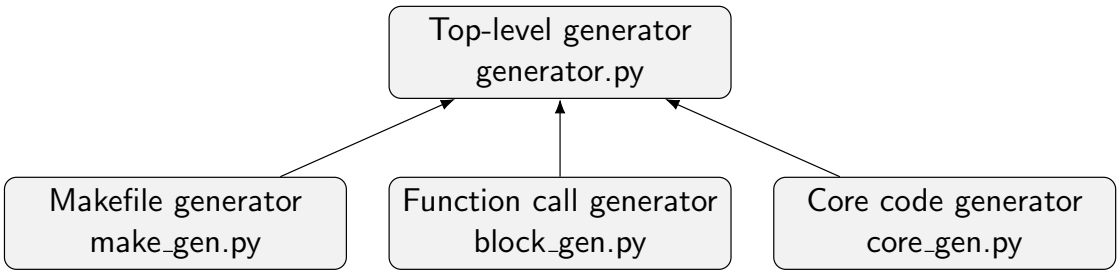

Figure 4.2: Code generator layout

Aside from that, the Makefile (`make_gen.py`) and core configuration file for the simulator are also generated, based on the number of cores required in the configuration, this is illustrated in Figure 4.3.

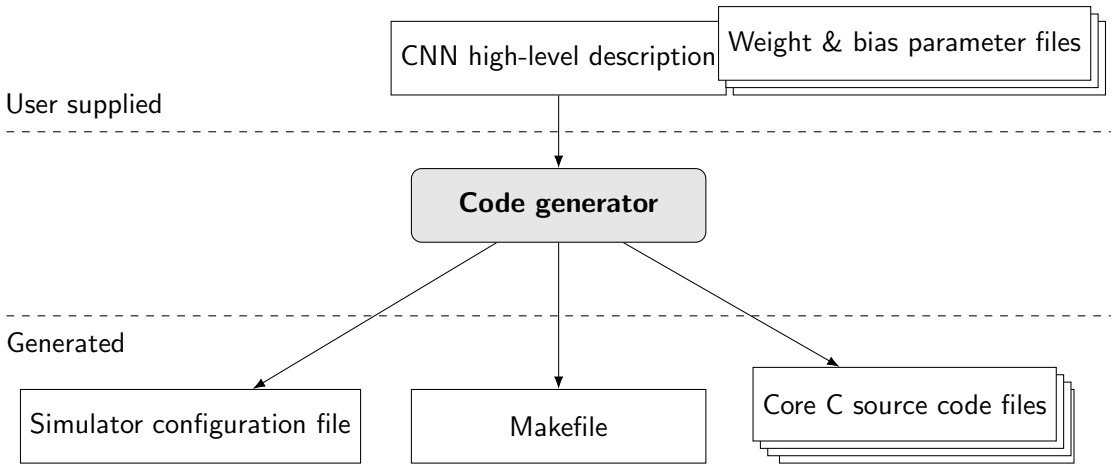

Figure 4.3: Input files and generated output files

In the high-level description the layers can be configured and assigned to a core. Each core can also be configured to run either with 1, 2 or 4 contexts in parallel. In this way

the DLP and ILP can be exploited on a layer-by-layer basis. An example of a high-level description for an MNIST network with 6 cores can be seen in Listing 4.3.

```
1  [core:4]
2  conv2d(5, 5, 1, 1, 28, 28, 0, 1, 20, params/layer_b0, params/layer_w0)
3
4  [core:1]
5  pool_max(2, 2, 2, 2, 24, 24, 0, 20)
6
7  [core:4]
8  conv2d(5, 5, 1, 1, 12, 12, 0, 20, 50, params/layer_b1, params/layer_w1)
9
10 [core:1]
11 pool_max(2, 2, 2, 2, 8, 8, 0, 50)
12
13 [core:4]
14 conv2d(1, 1, 1, 1, 1, 1, 0, 800, 500, params/layer_b2, params/layer_w2)
15 nonlinear_relu(1, 1, 500)
16
17 [core:4]
18 conv2d(1, 1, 1, 1, 1, 1, 0, 500, 10, params/layer_b3, params/layer_w3)
```

Listing 4.3: High-level description of a MNIST network

It can be seen that the convolution operations operate on four 2-issue contexts by the keyword `[core:4]`, while the pooling operations operate on one 8-issue context via the keyword `[core:1]`. The fully-connected layers (line 14, 18) are also modeled as a convolution operation with a 1×1 kernel, which is just a single multiply-accumulate operation.

The nonlinear operation in the fifth core is chained to the convolution operation in the same core to save resources and should not impact the throughput significantly due to its low computational complexity.

The convolution layers also specify the files from which the parameter values for the weights and biases can be loaded from. These files will be loaded into the RAM memory and a memory offset is automatically calculated and inserted into the code.

4.3.4 Operation

By running the script `generator.py` the description file will be parsed and all the code files will be generated. Subsequently, a make command can be executed and the code binary for each core will be built separately. The simulator configuration file is then used to load all the binaries and RAM image files containing the parameters into the simulator and the network can be executed on the specified number of cores.

This completes the implementation of the code generator, we now have implemented all the components to run CNNs on the ρ-VEX platform, the next step is to examine if this can be executed on hardware.

4.4 Hardware synthesis

In this section the hardware synthesis of the streaming pipeline is discussed. We first determine the accurate memory sizes required with the help of the simulator and then implement and synthesize the design of the streaming pipeline.

4.4.1 Verifying storage requirements through simulation

To determine if it is feasible to run the MNIST network on the FPGA with small memory sizes, this was first tested in the simulator. The minimal memory sizes were determined by experimentation and can successfully execute the MNIST network.

By minimizing the amount of memory while simultaneously verifying the network functionality, the required memory sizes were obtained. These values include the binary, required data memory, parameters and spacing. The memory layout is presented in Figure 4.4.

The spacing was experimentally determined and was found to be necessary for correct operation. A more efficient and accurate approach would be to include the parameters in C header files and compile them into the binary file to remove the need for arbitrary spacing. This was however not implemented due to time constraints.

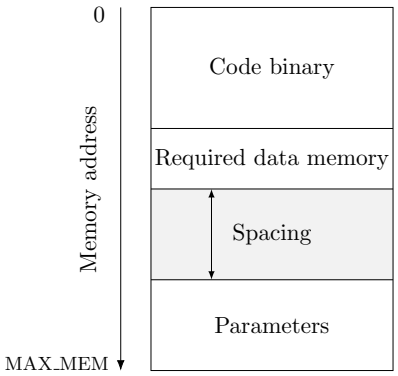

Figure 4.4: Memory layout of the core

In the simulator there is no explicit distinction between data memory and instruction memory, therefore we only determine one size value per core, the values being presented in Table 4.1.

Core	**Memory size**
Core 0	117 KiB
Core 1	140 KiB
Core 2	869 KiB
Core 3	98 KiB
Total	**1224 KiB**

Table 4.1: Minimal amount of storage required for successful execution in simulator

This is in line with the values from Table 3.3 and 3.4, however core 0, core 1 and core 3 require slightly more memory due to the required spacing, this should be taken into account when synthesizing and testing the design. We now proceed with implementation.

4.4.2 Implementation and synthesis

Implementing the design was done by changing the stream entity from a dynamic number of same-size cores to a fixed number of different sized cores. In our case we generate four cores in one stream with appropriate memory sizes. To implement the network on an FPGA, we determined in Table 3.3 and Table 3.4 that we need at least $1280 + 43 = 1323$ KiB of storage for parameters and buffers. Aside from this, 15 blocks of BRAM are required to implement the 2-issue ρ-VEX cores (see Figure 4.5).

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48E1	BU
ml605_streaming_platform		7/12052	16/12355	13/32464	0/408	0/417	0/16	
mnist_stream.cores		0/11897	0/12085	0/32109	0/384	0/417	0/16	
debug_demux_inst		8/8	1/1	21/21	0/0	0/0	0/0	
debug_stage_inst		17/17	100/100	6/6	0/0	0/0	0/0	
stream_gen[0].stream_...		0/11872	0/11984	0/32082	0/384	0/417	0/16	
debug_demux_inst		25/25	3/3	59/59	0/0	0/0	0/0	
debug_stage_inst		14/14	96/96	2/2	0/0	0/0	0/0	
stream_core_x0		0/2563	0/3049	0/6788	0/96	0/47	0/4	
core		67/2239	162/2836	153/5962	0/96	0/15	0/4	
data_bus_demu...		2/2	0/0	4/4	0/0	0/0	0/0	
debug_bus_dem...		23/23	2/2	74/74	0/0	0/0	0/0	

Figure 4.5: Module level utilization, core BRAM utilization is highlighted.

The streaming fabric was modified in such a way that there is a single stream in which each core is configured with different sized memories. Based on the findings from Section 4.4.1 the size of the data memory was slightly increased to accommodate the sizes from Table 4.1.

Our ML605 development board has an FPGA that has a maximum amount of 416 BRAM blocks of 36 kilobit size or in total 14976 kilobit (1828 KiB) of BRAM storage available. For a storage size of 1323 KiB and four ρ-VEX cores, we would require at least 301 BRAM blocks of 36 kilobit. It should therefore be possible to implement the stream. The clock speed was set at 200 MHz, which is the default for the ρ-VEX streaming fabric. The final design was synthesized with the sizes displayed in Table 4.2.

Core	**Instruction memory size**	**Data memory size**
Core 0	64 KiB	64 KiB
Core 1	128 KiB	16 KiB
Core 2	1024 KiB	4 KiB
Core 3	64 KiB	64 KiB

Table 4.2: Instruction memory and data memory sizes for each core

To access the memories, for instance for loading the parameters and binaries, a debug bus is implemented in the ρ-VEX streaming fabric. Based on the bus address, access to each memory of the core can be gained, as well as to the control registers. The default encoding of the bus address is as follows:

Address bits: $_{32}$ `---- SSSS IIII IIXX ---- ---- ---- ----` $_{0}$

This represents the full address in bits used to access the debug bus. The S bits select which stream should be accessed, the I bits select the core index within the stream and the X bits determine if the data memory ($0-$), instruction memory (10) or the control registers (11) should be accessed. The bits following the XX bits are used as the memory address to access the local memory. Because there are only 16 local address bits available we cannot access more than 64 KiB of instruction memory and not more than 128 KiB of data memory, because we can use an extra bit of the X flag in case of data memory.

Because we want to be able to access the entire instruction memory and the largest memory we synthesize is 1024 KiB, we need to adapt the debug address encoding in order to accommodate these sizes. This results in the following encoding:

Address bits: $_{32}$ `0SSS SIII IXX- ---- ---- ---- ---- ----` $_{0}$

In this case we can address only $2^4 = 16$ streams and $2^4 = 16$ cores, which for our purposes is sufficient. More importantly, we now have 21 bits available for local addressing, which can address up to 2048 KiB of instruction memory and 4096 KiB of data memory, enough for our reference implementation.

We summarize the synthesis results in Table 4.3, a full synthesis report can be found in Appendix A.

Resource	**Utilization**	**Utilization percentage of total available resources**
SliceReg	12.359	4%
SliceLUT	32.468	21%
BRAM	357	85%

Table 4.3: FPGA utilization of ML605 board

A four core stream is successfully synthesized with the required memory sizes with a clock speed of 200 MHz and an adapted debug bus. However, further work is needed to complete the hardware implementation as elements such as the FIFO registers have not yet been implemented. For larger networks the available memory resources are not sufficient to store all parameters and data, therefore a different solution would need to be found such as a memory bus connected to external DRAM.

4.5 Conclusion

In this chapter the implementation of the streaming ρ-VEX pipeline is presented. First we demonstrated a proof of concept to examine the viability of executing CNNs on the ρ-VEX platform.

An MNIST network was verified on both the simulator and FPGA single ρ-VEX core and a demonstration application was created to visualize the CNN inference on input data. After that, the implementation of the ρ-VEX pipeline was discussed. A multi-core simulator was implemented with a shared memory model and FIFO READ-WRITE

registers. The cores can access their own local memory as well as the local memory of the previous core in accordance with the ρ-VEX streaming fabric layout.

Following this, the code generator was presented that generates the code for each core based on a high-level description, as well as generating a Makefile and simulator configuration file. Finally, the hardware synthesis was discussed. Because the MNIST network is relatively simple, a four core design was successfully synthesized at 200 MHz, but more work is required to complete the implementation. Furthermore, larger networks cannot rely on local memories alone due to limited resources.

5 Evaluation

In Chapter 4 we discussed the implementation of the design. The different implementation details were presented with regard to simulator and hardware, as well as the required resources. After completing the implementation, in this chapter we evaluate the performance and accuracy of the CNN on the ρ-VEX streaming architecture. We also discuss possible optimizations based on the results, which could improve performance and resource usage.

5.1 Correctness verification

First, we verify if the results from CNN inference we obtained from the implementation are correct and compare the level of accuracy we observe to our reference network.

5.1.1 Methodology

Because no existing CNN implementation was available that could easily be ported to the ρ-VEX architecture, a subset of CNN operations has been implemented in the C programming language. This fact made it important to verify the correctness of each implemented operation before using it for evaluation or benchmarking.

An existing CNN is implemented with known parameters, input map and output values with the Caffe machine learning framework. First, a forward pass on the CNN with the known input values is performed and the output of the last layer is verified. Subsequently, the intermediate output values of each layer in the network are converted to 16-bit signed integers and exported from Caffe to a C header file format by means of the PyCaffe scripting interface.

The same CNN is then run on the ρ-VEX with our implementation and a forward pass is performed with the same input data and parameters. The output of each layer is then compared to the values in the header file. If any of the compared values are not equal, a mistake in the implementation is likely and should be investigated further.

5.1.2 Results

The output data of the test network (MNIST) of a single input sample matches the output of the Caffe version of the network quantized to 16-bits. The output of each layer was individually compared and found to be equal to the reference output.

To achieve the exact same output, most effort was spent on implementing a rounding function that would behave in the same exact way as the rounding in the reference implementation. By inspecting the Ristretto implementation code, an accurate rounding function could be implemented in such a way that the results correspond with each other.

Next, the network was executed on the ρ-VEX on 1.000 samples from the MNIST test set and achieved a 99% accuracy, matching the reference implementation within 0.3%. Due to the simulation time length, the entire test set of 10.000 samples was not verified, therefore the complete level of accuracy of the test set could not be confirmed.

5.1.3 Conclusion

We verified the implementation based on matching the output of the individual layers of a network with the reference implementation, which was found to be correct, and confirming the level of accuracy of 1.000 input samples within 0.3%. We therefore conclude that the implementation is valid for the tested network and continue with the performance measurements.

5.2 Performance measurements

In this section we examine the performance of the streaming pipeline for CNN inference. We analyze the throughput and latency of the network using the MNIST example. Subsequently, we examine the performance benefits of pipelined execution of the CNN inference. Finally, we measure the ILP of the implementation and investigate how this affects performance.

5.2.1 Methodology

To measure the performance of the implemented streaming architecture, the existing ρ-VEX simulator is used. This simulator has close to cycle-accurate counters for executed cycles, bundles and instructions, which makes it suitable for performance measurements.

There is also the option to export execution traces of the assembly code to external files for further analysis. The execution traces will be used to analyze the ILP of the implementation.

To measure the performance, we implement the LeNet-5 inspired network with six layers, based on the Caffe reference implementation (see Figure 5.1). This network was trained with a level of accuracy of 99.14% on the MNIST dataset. Each layer of the CNN has its own dedicated core for processing, this enables pipelining of the input samples, which could result in a higher throughput. We assume a clock speed of 200 MHz, as this was the frequency we achieved on the hardware synthesis in Section 4.4.

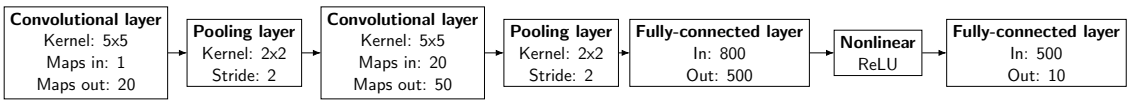

Figure 5.1: LeNet-5 [8] inspired network for MNIST dataset.

We start with measuring throughput and latency and analyze how multiple contexts and pipelining can improve the performance.

5.2.2 Throughput and latency measurements

Measuring the throughput and latency, we observe several factors involved. By varying the number of independent contexts in each ρ-VEX core, more performance can be gained by taking advantage of the data parallelism present in the CNN layers.

Configuration	**Latency of single sample**	**Throughput**	**Latency improvement**	**Throughput improvement**
Single context of each core	567 ms	3.08 samples/s	1.0×	1.0×
Core 0, 2: two contexts	401 ms	4.34 samples/s	1.41×	1.41×
Core 0, 2: four contexts	335 ms	4.47 samples/s	1.69×	1.45×
Core 0, 2, 4, 5: four contexts	187 ms	9.65 samples/s	3.03×	3.14×

Table 5.1: Latency and throughput measurements with multiple contexts

In Table 5.1 the results can be observed. It can be seen that adding contexts to the convolutional and fully connected layers improves both latency and throughput up to 3.14× by taking advantage of the data parallelism.

Benefits from pipelining

We now investigate the benefits of creating the pipeline on the performance of the CNN. The performance was measured by executing the network on different core configurations with ten input samples. Only single-context 8-issue cores were used, due to the inability to run multiple contexts in every configuration. This limitation is further explained in Section 5.5.3. In Table 5.2 we can see the results of testing the network on a different numbers of cores, one to six.

It can be seen that for ten samples, we can get an improvement up to 1.77×. It can also be seen that most improvement is gained by assigning convolution operations to separate cores. This is in line with the observation from 2.6.1 that most computation of the CNN takes place in the convolutional layers.

We verify the results by running the network with the same core configurations on a single sample and checking the resulting performance, which is displayed in Table 5.3. We can see there is no benefit from multiple cores in the single sample case, therefore the performance increase in Table 5.2 is strictly due to the increased throughput achieved by pipelining multiple samples. In Section 5.5.3 we go into more detail about how we can maximize the pipelining performance while minimizing the number of cores to reduce the amount of resources.

Number of cores	Total cycles	Improvement	Description
1	1132362318	1.0×	Baseline
2	745759628	1.52×	One core with all convolution operations, one with all fully-connected layers
3	644038957	1.76×	Convolutions spread over two cores and one core with fully-connected layers
4	644048474	1.76×	Convolutions spread over two cores, fully-connected spread over two cores
6	641549277	1.77×	Convolutions and pooling on individual cores, fully-connected spread over two cores

Table 5.2: Pipeline performance from ten samples.

Number of cores	Total cycles	Improvement	Description
1	113293359	1.0×	Baseline
2	113305985	1.0×	One core with all convolution operations, one with all fully-connected layers
3	113340487	1.0×	Convolutions spread over two cores and one core with fully-connected layers
4	113348276	1.0×	Convolutions spread over two cores, fully-connected spread over two cores
6	113536557	1.0×	Convolutions and pooling on individual cores, fully-connected spread over two cores

Table 5.3: Performance with one sample.

5.2.3 ILP measurements

Next, we measure the ILP of the MNIST network by analyzing the operations of each layer individually, which should give an indication what the range of ILP is. This is done by executing each operation separately on the ρ-VEX and analyzing the execution trace of a number of iterations to obtain the average ILP. Measuring the ILP of the MNIST network the following results are obtained in Table 5.4.

MNIST ILP					
Conv1	**Pool1**	**Conv2**	**Pool2**	**IP1**	**IP2**
1.51	1.94	1.56	1.94	1.45	1.59

Table 5.4: ILP for operations of MNIST layers

It can be seen that the ILP is between 1.45 and 1.94, the convolution and full connected layers are around 1.5 while the pooling operations are around 1.94. The lower ILP for the convolution and fully connected layers can be explained by the fact they have more data dependencies than the pooling operations because of the required parameters, which makes it more difficult to schedule the instructions efficiently in a VLIW architecture. From these results it can be seen that all operations would benefit from being divided over four 2-issue cores as no operation has an ILP higher than 2.

To obtain more data on the convolution operation, we also measure the ILP for varying convolution sizes to see if this would influence the ILP, this can be seen in Table 5.5.

$\mathbf{1 \times 1}$	$\mathbf{2 \times 2}$	$\mathbf{3 \times 3}$	$\mathbf{5 \times 5}$	$\mathbf{10 \times 10}$
1.52	1.51	1.49	1.51	1.51

Table 5.5: ILP for varying convolution kernel sizes

We observe that the ILP remains quite constant for the most common kernel sizes of CNNs, again most likely due to the data dependencies preventing any possible ILP gains in higher kernel sizes. We conclude that taking advantage of DLP through multiple contexts with small issue-width will yield better performance than less contexts with wide issue-width because of the low ILP. However, the ILP could be improved by optimizations such as loop unrolling, which is investigated in Section 5.5.

5.2.4 Conclusion

From the results we conclude that the throughput and latency can be improved by dividing the data of a single operation over multiple contexts, taking advantage of the data level parallelism in the convolutions. We achieved an improvement up to 3.03× in latency and 3.14× in throughput for the given network.

By dividing the network operations over multiple cores, we can take advantage of pipelining and achieved a 1.77× improvement in throughput of the network.

The ILP of the measured operations was between 1.45 and 1.94, which is too low to take advantage of an issue-width wider than two. This also makes the argument for using multiple contexts. However, ILP could be increased by optimization techniques. We will now continue with discussing different optimizations in the following sections.

5.3 Optimizations

To extract more performance out of the CNN streaming architecture on the ρ-VEX platform, different optimization strategies are possible. In this section several of these optimizations are examined and we analyze the amount of performance gain they can achieve.

For clarity they have been divided into two categories: hardware and software optimizations. Hardware optimizations imply changes to either the ρ-VEX processor, the interconnect between the processors or the memory architecture. Software optimizations

are optimizations for the software implementation of the CNN operations and only imply changes to the programming code. Some optimizations have both a hardware and software component such as the SIMD vector unit, but these will be discussed as hardware optimizations.

We have seen that the convolution operation 2.6.1 is the dominant operation in the computation time of a forward pass of a CNN. For this reason, most of the discussed optimizations will be focused on improving the performance of the convolution operation. This is because any performance gain in this operation will have the most impact on the CNN computation time as a whole.

5.4 Hardware optimizations

Hardware optimizations are possible to improve the latency and throughput of CNNs on the ρ-VEX platform. Performing the convolution operation consists of mostly MAC operations, so we focus on increasing the number of MAC operations that can be executed per cycle on the ρ-VEX core. There are several possibilities for this that we will discuss here.

5.4.1 Dedicated multiply-accumulate instruction

To save cycles during computation of CNN inference, a combined multiply-accumulate instruction could be beneficial due to the large number of MAC operations. We will therefore analyze if such an instruction could be beneficial in reducing the number of cycles.

Issue addressed

In [66, pp. 249-251] it is shown that we can create combined multiply-accumulate units instead of a separate multiplier and adder. In this way the expression $y = (ax) + b$ can be calculated in a single unit. Because the convolution operation mainly consists of multiply-accumulate operations using a running sum, we can benefit from removing the explicit addition operation from the instruction stream and reduce the code size and latency. This can be achieved by implementing a dedicated multiply-accumulate instruction that combines these operations in a single computational unit.

Approach

To determine possible performance benefits of implementing a multiply-accumulate instruction, we will analyze the execution trace of a single MAC operation. When we compile code that evaluates the expression $y = (ax) + b$ for the values $a = \texttt{0xF}$, $b = \texttt{0xC}$ and $x = \texttt{0xD}$, we obtain the execution trace on ρ-VEX in Listing 5.1.

```
1 mpyllu r25(0xc3) = r25(0xf), r26(0xd)
2 nop;;
3 nop
4 nop;;
5 add r24(0xcf) = r24(0xc), r25(0xc3)
6 nop;;
```

Listing 5.1: Separate multiply and addition instructions

What can be seen is that after registers have been loaded with the values, the multiplication operation is scheduled in its own bundle and that another empty bundle follows it because of the 2-cycle multiplication instruction boundary constraint of the ρ-VEX. Only then can the addition instruction take place.

Therefore, the expression takes 3 cycles in total to complete without memory operations. If we combine these instructions in a multiply-accumulate operation, this could take 2 cycles including the empty bundle constraint, which is shown in Listing 5.2.

```
1 mpyllu_add r24(0xcf) = r25(0xf), r26(0xd), r24(0xc)
2 nop;;
3 nop
4 nop;;
```

Listing 5.2: Combined multiply-accumulate instruction

The only problem is that the ρ-VEX opcodes are too short to accommodate three register identifiers. It could be solved by assigning an existing fixed register as accumulator register. However, this would require more read ports on the register file than are currently available, which currently is already a complex issue due to the fact that the register file needs to be accessible from every pipelane [9].

A simpler alternative would be to create a dedicated accumulation register inside the multiply-accumulate unit. Subsequent accumulations would be stored here, but this would then require instruction support to move accumulation register values to and from the regular register file.

Expected gains

A 5×5 convolution would require 25 MAC operations to be performed. Assuming the ideal case in which all required values are present inside the register file, this would take $25 \cdot 3 = 75$ cycles to complete. When implementing our multiply-accumulate operation this would only take $25 \cdot 2 = 50$ cycles, a speedup of $1.5\times$.

In the worst case scenario where we load all values from memory and then perform a store of the accumulated value, this would take 8 cycles while ignoring memory latency, with MAC instruction it would take 7 cycles. For a 5×5 convolution this yields $25\cdot8 = 200$ cycles vs. $25 \cdot 7 = 175$ cycles, a speedup of $1.14\times$.

Because we need a way to store the intermediate value, we implement an accumulator register inside the MAC unit and add `LoadAccumulator` and `StoreAccumulator` single-cycle instructions that can move values between the register file and the accumulator.

When taking these added operations into account, a speedup between 1.10× and 1.48× can be achieved for convolution sizes between 3×3 and 9×9.

Implementation cost

The simplest way to implement the multiply-accumulate instruction in the existing ρ-VEX architecture would be to have a drop-in replacement for the current multiplication unit.

When the ρ-VEX is used as a softcore on a FPGA platform, generally the underlying multiplier is synthesized automatically depending on the targeted FPGA. When an FPGA has support for hardware multiplier blocks inside Digital Signal Processing (DSP) slices, these will be used as the underlying multiplier hardware. For example, the ML605 platform uses a Virtex-6 FPGA that contains 25×18 bit multiply-accumulate blocks [67]. Several of these blocks will then be cascaded to form our 32×16 multiplier inside the ρ-VEX but the exact configuration depends on the synthesizer.

These hardware blocks have superior performance over a manual implementation, this is because the manual implementation would have to run on top of the FPGA fabric, which by design is less efficient than a built-in ASIC implementation.

From the FPGA documentation we learn that the FPGA DSP slices already support the MAC operation, therefore implementing this operation would yield no extra hardware cost with respect to computational units. However, extra logic is required to implement the accumulator functionality and associated instructions.

5.4.2 SIMD vector processing unit replacing multiplication unit

To improve the number of MAC instructions executed per cycle, an SIMD vector unit could be implemented to operate on vector data instead of scalar data.

Issue addressed

The default configuration of a ρ-VEX processor has multiple execution lanes with an issue-width of 2 that can be combined to form a higher issue VLIW processor. Each lane contains a multiplier unit to multiply scalar values.

Because we want to optimize performance specifically for CNNs, the multiplication unit in some lanes could be replaced by a SIMD vector processing unit. For example a 5×5 convolution operation requires 25 MAC operations. By using 80-bit vector registers containing five 16-bit values, we can perform five MAC operations simultaneously in the same time it would take to perform one MAC operation on scalar values. Because convolution operations have high data-level-parallelism and no branching, the SIMD approach is ideal to increase performance.

The benefit of this approach is a reduced code size because less instructions are required for the same workload. It reduces the instruction fetch and decode bandwidth required to manage an instruction stream across many ALUs. Furthermore, memory access patterns are more predictable because a fixed number of aligned values can be loaded into the vector registers directly, which can be exploited by using burst reads on a memory interface connected to external memory.

Approach

To operate on multiple data in parallel, more demands are placed on the load/store unit and required bandwidth because we want to load more values simultaneously. When considering our size five vector unit again, we see that we need to load five values before we can execute the vector instruction. However, if we make sure that all values are aligned in banked memory, we see the setup latency to main memory only once for the entire vector. In any case this would require changes to either the existing load/store unit or creation of a dedicated vector load/store unit, which makes implementation more complex [30, p. 265].

Related work

When examining potential performance gains from SIMD, the Tarantula vector unit [68] is an extension to the EV8 (Alpha 21464) core, adding 45 vector instructions and 32 vector registers that each hold 128 64-bit values. Compared to the EV8 core, the speedup achieved across several benchmarks averaged at 5× with a peak speedup of 8×.

Analyzing a recent VLIW processor with SIMD capabilities, an example is the Hexagon V5 DSP [69]. This is a 4-issue VLIW processor with dual load/store execution lanes and dual SIMD execution lanes and has support for hardware multithreading. The two SIMD units are identical and support 64-bit values. It can perform regular ALU operations on single 8-, 16-, 32- and 64-bit integers as well as perform four 16×16 multiplications, two 32×16 multiplications or one 32×32 multiplication operation.

A similar approach could be taken to implement SIMD in the ρ-VEX core while maintaining the existing multiplier so that no functionality is lost. It would come at the cost of a more complex SIMD design, because it now needs to support single multiplications as well as operations on multiple values. By taking the approach to not sacrifice the multiplier could however lead to a higher ILP, because a more diverse set of instructions can be scheduled in the same execution lanes. Furthermore, vector instructions have less dependencies on other instructions because they operate on vector registers and can therefore be scheduled more flexibly.

Expected gains

When analyzing the performance gains of a MAC SIMD operation, we look at different vector lengths. We assume a DDR3 memory interface with a burst length of 8 and word size of 64 bit. The datatype is 16-bit signed fixed point. We assume that before each burst, we see a 12 cycle setup latency (default value from the ρ-VEX simulator).

If we assume we can perform all MAC operations in parallel in 2 cycles, the memory becomes the main contributor to the latency. If we assume that after the initial memory latency, we can read 64 bit each cycle for a burst length of 8, we can read 32 16-bit values in 19 cycles including initial latency.

We compare it to the regular sequential MAC with all values in registers and with all values accessed through memory but with the best-case scenario access latency of one cycle.

Vector size	Sequential	Sequential + load	Vector instruction	Speedup seq.	Speedup seq. + load
2	4	8	14	0.29×	0.57×
4	8	16	15	0.53×	1.07×
8	16	32	17	0.94×	1.89×
16	32	64	21	1.52×	3.04×
32	64	128	40	1.6×	3.2×
64	128	256	78	1.64×	3.28×

Table 5.6: Analysis of possible vector speedup compared to sequential MAC operations

In Table 5.6 we can see that for a vector size of 16, the SIMD unit begins to yield good results compared to both sequential cases. For increased sizes the speedup begins to plateau between 1.6× and 3.3× depending on the sequential scheme. This is because from 32 elements onward, we need to perform a new burst read from memory that adds more setup latency. A vector size between 16 and 32 elements in this case would yield the best results.

Implementation cost

To estimate the area used by the SIMD unit, we analyze the current utilization of the multiplier unit in a single ρ-VEX pipelane, presented in Table 5.7. It was obtained from the module level utilization of the synthesis report of an 8-issue ρ-VEX core, which is included in Appendix B.

SliceReg	LUT	DSP
33	41	2

Table 5.7: Multiplier utilization single pipelane on ML605 FPGA board

To get a rough estimate of the SIMD utilization, we add a multiplier unit to the pipelane for each added vector element. This creates the estimated utilization in Table 5.8.

We can observe that for even large vector sizes, the utilization is acceptable compared to the resources used by the entire pipelane. A vector length of 32 would create an average increase in LUT/slices of 11%, while the 400% increase in DSP blocks has to be seen in perspective, because we started with only 2 DSP blocks. Based on these numbers and the available FPGA resources, an SIMD unit with vector size 16 or 32 is feasible.

Because we want to operate on vector registers separate from the existing general purpose registers, an extra register file will need to be implemented. Ideally, we would like to access these registers from each pipelane in the ρ-VEX processor in case we implement an SIMD unit in each pair of pipelanes. As shown in [9] the ρ-VEX requires 16 read ports and 8 write ports to the register file in order to access registers from each pipelane. It is implemented on FPGA architectures by duplicating the BRAM blocks to

Vector size	**2**	**4**	**8**	**16**	**32**	**64**
SliceReg	66	132	264	528	1056	2112
LUT	82	164	328	656	1312	2624
DSP	4	8	16	32	64	128
As percentage of current single pipelane utilization						
SliceReg	0.7%	1.5%	2.9%	5.8%	11.7%	23.4%
LUT	0.5%	1%	2.0%	4%	7.9%	15.9%
DSP	25%	50%	100%	200%	400%	800%

Table 5.8: Utilization estimate of SIMD unit

facilitate more parallel read/write ports and keeping the most recent written to BRAM in a live value table. For the existing general purpose register file of 64 32-bit registers, this yields the utilization given in Table 5.9.

SliceReg	**LUT**	**RAMB18E1**
1392	15591	128

Table 5.9: General purpose register file utilization of ρ-VEX (from [9])

To obtain a rough estimate of the size of the vector register file, these metrics can be scaled to the size and number of vector registers required. Vector lengths of 16 and 32 will be considered. For 32 values of 16-bit, 512-bit size registers will be required. For 16 values of 16-bit, 256-bit registers will be required. To limit resource usage, only 16 registers will be implemented. The general purpose register requires $32 \cdot 64 = 2048$ bits of storage. Combined with the required read/write ports and multi-context support, this yields a usage of 128 RAMB18E1 blocks.

For sixteen 256-bit and 512-bit registers, 4096 and 8192 bits are required, respectively. This is double and quadruple the size of the original register file. By scaling the resources in a similar way, the results in Table 5.10 are obtained.

	SliceReg	**LUT**	**RAMB18E1**
16 wide	2784	31182	256
32 wide	5568	62364	512
Increase percentage over core			
16 wide	16,5%	77,3%	191,0%
32 wide	33,0%	154,7%	382,1%

Table 5.10: Estimate of vector register file utilization

It can be observed that there is a significant increase in utilization in both LUT and BRAM blocks. Even though for the 32-wide case, the amount of resources used still falls within the available resources of the ML605 FPGA development board.

However, we must take into account an increase in control logic due to the implementation of new vector instructions and modifications to the load/store unit, which will

increase the utilization further.

Given the speedup numbers and the amount of resources used, it can be concluded that going from 16-wide to 32-wide vectors would double the amount of required resources. However, from Table 5.6 can be determined that the gain in performance is around 5%. Therefore, the increase in resources is not justified and a 16-wide vector would be the best choice.

	Slice Registers	**LUT**	**DSP48E1**	**RAMB18E1**
Multipliers	528	656	32	0
Register file	2784	31182	0	256
Total	3312	31838	32	256
Percentage of core	19,6%	79,0%	200,0%	191,0%

Table 5.11: Total estimate of 16-wide SIMD unit with sixteen 256-bit registers.

In Table 5.11 the total estimate of a 16-wide SIMD unit is given. It can be observed that the increase of slice registers is around 20%, but the amount of LUT, DSP and BRAM blocks increase significantly, an increase between 79% and 200%. These values are still within the limits of the ML605 FPGA board and would therefore be synthesizable. However, care must be taken, because this estimate does not take into account extra control logic and modifications to the memory unit. According to Table 5.6 this would yield a theoretical speedup between 1.52× and 3.04×.

One drawback of this proposal is that it is not possible to implement more than one SIMD lane in a ρ-VEX core due to resource constraints of the FPGA. In that case it would be necessary to reduce the vector size of the unit and/or the number of registers.

A possibility would be to implement a smaller vector unit, which compensates for reduced speedup by executing all SIMD lanes in parallel. It would however require analysis of memory bandwidth and memory bus communication, due to multiple memory burst requests being issued in parallel.

5.4.3 Multidimensional SIMD with polymorphic registers

When implementing an SIMD vector processing unit to perform more MAC operations per cycle, we are constrained to a fixed vector size. For vectors smaller than the vector unit size, resources will be wasted because we cannot utilize the hardware fully in that cycle. Furthermore, the convolution operation operates on two-dimensional input data that is independent, therefore, in theory, we would like to have one SIMD instruction to perform an entire convolution instead of separate MAC operations to reduce the code size further. It would however mean all data should be already present in a register file and have the correct dimensions to maximize hardware utilization.

Approach

In [70] Ciobanu et al. propose a polymorphic register file with a two-dimensional layout. This register file allows the dynamic creation of multidimensional registers of arbitrary

sizes. The rationale for this design being that performing operations on 2D matrices with scalar instructions requires a significant number of committed instructions.

Using 1D or even 2D register files could reduce the number of required instructions by utilizing the SIMD principle that reduces the static code size. To improve this further, register file utilization can be optimized by making the register sizes dynamically configurable during runtime, supporting multiple dimensions simultaneously.

Expected gain

The achieved speedup for a dynamic programming problem is 2.7× compared to the Cell Synergistic Processing Unit (SPU) with similar throughput. For a sparse matrix vector multiplication, a speedup of 7× can be achieved compared to a scalar PowerPC (PPC) processor when using the 2D register to store multiple rows of the matrix and process them simultaneously.

In [71] Ciobanu and Gaydadjiev demonstrate that a separable 2D convolution operation can be performed on the Polymorphic Register File (PRF). It achieves better performance than an NVIDIA c2050 GPU on convolution dimensions of size 9×9 and upwards, which have sufficient computational expense to favor the PRF.

For smaller separable 2D convolutions the PRF is limited by the bandwidth to the register file and cannot outperform the GPU. For a 33×33 window a speedup of 4× can be achieved by the PRF with 4 vector lanes.

Implementation details

The Polymorphic Register File could be beneficial for increasing the performance of two-dimensional convolution operations in CNNs. Most CNN convolution operations operate on small window sizes such as 3×3 and 5×5. These windows can fit into a two dimensional register file of limited dimensions. Both the parameters and the input data window can be stored in the register file and a convolution operation can be performed directly by a special instruction (equivalent to a one-dimensional SIMD MAC instruction). By changing the register dimensions when performing different types of convolutions, the register file can be utilized as efficiently as possible. For instance, the register file could fit more 3×3 windows than 5×5 windows by changing the register dimensions. However, attention must be paid to the memory bandwidth bottleneck for small convolutions as was indicated in the previous section.

Given that a speedup on a large 2D convolution can be achieved compared to a GPU architecture and a speedup compared to the Cell SPU with SIMD support, it could be beneficial to implement a polymorphic register file to use for convolution computation, but it will require a more detailed investigation.

5.5 Software optimizations

Unoptimized code can be written or compiled in such a way that it cannot take full advantage of the underlying hardware. By taking into account the details of the hardware, the code can be optimized in a way that increases performance. In this section we

explore a number of these optimizations.

5.5.1 Loop unrolling

Issue addressed

Our convolution implementation is implemented as four nested loops, one loop over the input feature maps, one over the output feature maps and two loops over the convolution kernel dimensions (see Listing 3.1). It has been shown [30, pp. 157-161] that loop unrolling can improve performance because it reduces the branch penalties and arithmetic with respect to the loop index. It improves data independence of instructions, which can increase the possible ILP. Higher ILP is especially beneficial for wide VLIW architectures such as the ρ-VEX.

Approach

To measure any possible performance gains from loop unrolling, we measure the number of cycles for a single convolution with different sizes. We then compare the baseline performance with a version with fixed loop variables, several unroll factors and a manual unrolled version to discover possible performance gains. For several convolution sizes we compare manual unrolling with automatic unrolling with `#pragma` statements to observe any differences.

Expected gains

Kernel size (28×28 data)	**Fixed variable loop gain**	**Unroll gain**	**Unroll factor**	**Manual Unroll**
1×1	1.04×	1.04×	2	1.03×
3×3	1.13×	1.16×	4	1.13×
5×5	1.11×	1.16×	6	1.11×
7×7	1.14×	1.30×	8	1.24×
9×9	1.09×	1.25×	10	1.19×

Table 5.12: Summary of gains from loop unrolling

In Table 5.12 we can see the results for the performance improvement of loop unrolling with different kernel dimensions. An improvement between 1.04× and 1.30× can be achieved by adding a `#pragma` statement to the convolution loops with the indicated unroll factor. Only even unroll factors were measured but an unroll factor higher than the loop iterations has no effect beyond that limit. It can also be seen that using a `#pragma` statement gives marginally better performance than manually unrolling the loops.

5.5.2 BLAS routines for ρ-VEX

Issue addressed

The method of implementing convolutions by means of nested loops in code is not optimal for fast computation on modern CPUs. This situation can be improved by reorganizing the data in a more regular structure that can take advantage of the features of modern processors.

Current deep learning frameworks use linear algebra routines in their computations, such as matrix-matrix and matrix-vector products. To improve the performance of these operations, special libraries were developed that optimize numerical linear algebra routines. Originally, these libraries were meant to improve the performance of scientific computations and are called Basic Linear Algebra Subprograms (BLAS).

These routines are heavily optimized by hardware vendors and the academic community in order to get the highest possible performance out of different platforms, some examples are Intel MKL [45] and OpenBLAS [72]. By mapping the convolution operation of a CNN to a matrix-matrix multiplication, we can take advantage of optimized BLAS routines to improve performance. However, because the ρ-VEX platform does not have default floating point support yet, such routines would have to be implemented for fixed-point datatypes. Such an example can be found in [73], which are optimized matrix multiplication routines for low precision fixed-point datatypes.

Approach

To take advantage of BLAS routines, first the convolution problem will need to be mapped to matrices. In [74] Chellapilla et al. show how a convolution operation can be unfolded to a matrix-matrix product. The input data is unrolled in such a way that each row of the input matrix contains all the input values necessary to compute one output element. Similarly, the weight kernels are also unrolled and combined to form the kernel matrix. This does imply some duplication of data, but the cost of this is negligible compared to the total number of operations for the matrix product.

The convolution can then be calculated by performing a matrix-matrix product between the input matrix and kernel matrix and can now be mapped to efficient BLAS routines.

Expected gain

By unrolling the convolution and using Intel MKL BLAS routines on a CPU, Chellapilla et al. achieve a speedup between 2.42× and 3.00× on the MNIST and LATIN datasets when performing 1.000 forward and 1.000 backward passes.

5.5.3 Layer grouping to reduce resource usage

Issue addressed

At the moment, when we run each layer of the CNN on its own dedicated core in the stream, we can increase the throughput by pipelining. We have seen in Section 5.2.2

that an increase in throughput of up to 1.77× can be achieved by this method. Because some computations such as max-pooling take significantly less time compared to the convolutions, cores that execute these layers are frequently idle. It would be more efficient if the operations that are computationally less intensive could be combined with other operations on a single core, this would reduce the number of cores while maintaining similar performance.

Approach

The code generation is adapted in such a way that multiple operations can be chained together on a core. When using multiple contexts however, synchronization between the contexts will be required in case an operation uses all the output maps of the previous layer as input data.

Because we lack the synchronization primitives required to do this effectively, currently chaining operations is limited to either single context cases or between operations that operate on the same subset of input and output data so that synchronization is not required.

In practice this means we can chain convolution operations followed by pooling and nonlinear operations when using multiple contexts, but not pooling operations followed by convolution operations, because the convolution operates on all the input maps. We should however be able to demonstrate some resource reductions with this limited capability.

Expected gain

We would expect to see no significant loss of performance when grouping computationally less intensive layers with convolution operations. It could drastically reduce the amount of resources used for large networks, as they usually have a sequence of convolution-pooling layer pairs.

Measured gain

We measured the performance of several core configurations on a sequence of ten input samples and present them in Table 5.13.

Cores	Cycles	Improvement	Description
6	641549277	1.00×	Baseline
5	641397541	1.00×	Merge first conv. and pool. layer
4	644048474	0.99×	Merge first and second conv. and pool. layer
3	745769151	0.86×	Merge first four layers
2	1125844883	0.57×	Merge first five layers
1	1132362318	0.56×	Merge all layers

Table 5.13: Analysis of layer grouping for the MNIST network over ten input samples

It can be observed that merging the first two convolution and pooling layers has little impact on the total number of cycles. The non-linear layer is already merged with

the first fully-connected layer in the baseline due to its simplicity. If we start merging convolution operations, performance begins to drop because we lose the advantages of pipelining, therefore this is not beneficial.

We can conclude that chaining computationally less intensive operations to convolution operations has little effect on the performance and has the possibility for reduced resource usage. In our example a resource reduction of 1.5× can be achieved with little performance reduction.

5.5.4 Lower precision calculations

Issue addressed

Much attention is being paid to reducing the datatype precision required to perform CNN calculations, while maintaining the same level of accuracy in classification problems [21] [75] [76] [77].

Because an embedded processor has limited resources and sometimes lacks a floating point unit, it would be beneficial for these types of platforms to reduce the datatype precision as much as possible without sacrificing performance. We already use 16-bit fixed point datatypes for our CNN inference, but we will investigate if reducing this further would create higher performance.

Approach

We already switched to 16-bit fixed point values because this would yield 32-bit multiplication results that fit directly into the ρ-VEX architecture and in theory reduce any overhead caused by processing higher bit-width products.

Another reason was that for CNNs we have seen that 16-bit fixed point is a good tradeoff point for achieving a high level of accuracy while avoiding the use of floating point arithmetic.

To examine if there is any difference in performance when varying datatype precision, we will create three implementations of convolutions for 8-bit, 16-bit and 32-bit fixed point values and compare the execution time for each of these operations. We remove any rounding operations to get a clearer picture of computational expense of only the convolution arithmetic.

Expected gain

We already explained our reasoning for using 16-bit fixed point because of the multiplication result size, even though the ρ-VEX operates on 32-bit values. Considering 8-bit values, intuitively it would seem there would be no gain in performance compared to 16-bit, because there is no advantage in instruction cycles or memory latency. We therefore expect to see only a slight improvement in performance for 16-bit values compared to 32-bit and no improvement between 16-bit and 8-bit values.

Measured gain

In Figure 5.2 the speedup for several convolution sizes and precisions is presented. We use 16-bit as our baseline precision and compare the other precisions to it. There is a little benefit of using 8-bit values with almost equal performance to 16-bit, which implies the overhead of using precisions less than 32-bit is not strongly related to the actual precision width.

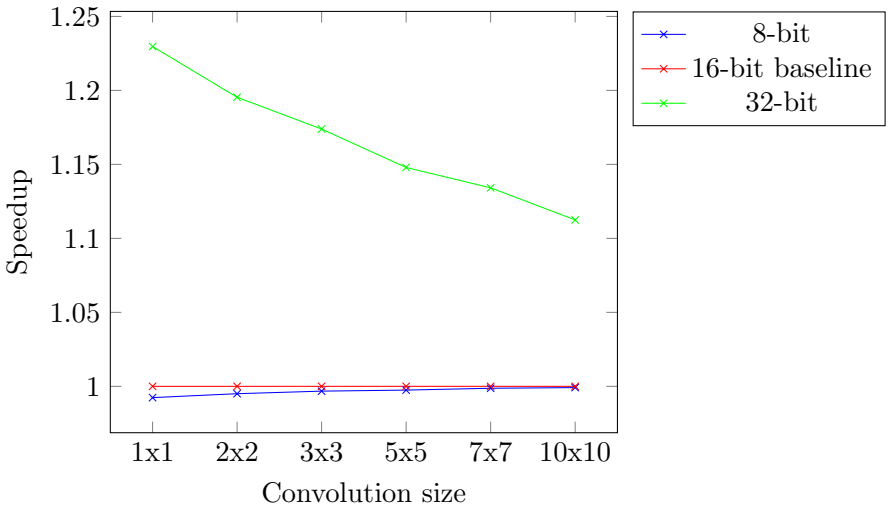

Figure 5.2: Comparing precisions to 16-bit

The more interesting observation is that using values with 32-bit precision outperforms both 8-bit and 16-bit precision with an improvement between 1.11× and 1.23×. The overhead of manipulating lower precision values in the 32-bit ρ-VEX architecture is of such a degree that no benefits can be gained from switching to lower precision, not even with a multiplication-dense operation such as a convolution.

The only side note here is that we assume there are no large discrepancies in the way the compiler schedules these operations for different datatype precisions. To get a clearer picture it would be necessary to manually write these operations in assembler code to find any possible areas of increasing performance of lower precision values, which would also tie into the development of the BLAS routines discussed in Section 5.5.2.

5.6 Conclusion

In this chapter we first evaluated the correctness and performance of the CNN implementation on the ρ-VEX platform. The correctness was verified on a single input sample and was observed to match the Caffe reference implementation. Next, we tested 1.000 samples from the MNIST test set and achieved a level of accuracy within 0.3% of the reference implementation.

The performance of the implementation was subsequently analyzed. Based on measurements, we determined that using multiple contexts on convolutional and fully connected layers improved both latency and throughput with a factor between 1.4× and

3.14×. We then analyzed the possible gains from pipelining and achieved a 1.77× improvement in throughput.

Next, the ILP of the implementation was analyzed and we found a value between 1.45 and 1.94. This explains the benefit from multiple contexts we have seen, as a wide VLIW core did not increase performance in this case. It is therefore more useful to exploit the DLP using multiple contexts in parallel.

After the evaluation we analyzed several possible optimizations to our CNN implementation on the ρ-VEX platform. We examined both hardware and software optimizations and obtained possible gains based on analysis or measurements. The optimizations are aimed mostly at improving the performance of the convolution operation as this is the most computationally complex part of the CNN inference.

For the hardware side, we first analyzed the implementation of a dedicated Multiply-Accumulate instruction in the ρ-VEX core. This could yield an expected performance gain of between 1.10× and 1.48×.

Next, the implementation of vector extensions was investigated as well as possible gains. This resulted in an expected gain of between 1.52× and 3.04× for a 16-wide vector unit, while keeping increased resources within the limits of the ML605 FPGA development board.

This was followed by a review of a Polymorphic Register File from literature as a method for performing two-dimensional SIMD operations, the motivation being to reduce static code size. A gain was reported on similar problems such as a separable 2D convolution. However, to make an accurate prediction on possible performance increases, further investigation is required.

On the software side we analyzed the loop unrolling technique as a possible performance improvement, because the convolution operation was implemented as nested loops. A gain was measured of between 1.04× and 1.30× for different convolution sizes using the compiler directives for unrolling.

We then examined the possibility of implementing dedicated BLAS routines for the ρ-VEX architecture to speed up matrix multiplication operations. This is motivated by the fact that a convolution operation can be mapped to a matrix multiplication to take advantage of such optimized routines. Based on literature results, this could yield an expected gain of between 2.42× and 3.00×.

To reduce resource usage, grouping of CNN layers was investigated. This was motivated by the fact that running layers with low computational complexity on a dedicated core does not yield enough performance benefits to justify the extra resources. By combining these layers with layers with high computational complexity, resource usage can be reduced. In our example we measured a 1.5× reduction in resources with a negligible performance loss.

Finally, we analyzed the impact of changing datatype precision to improve performance. We tested 8-bit, 16-bit and 32-bit datatypes on a variety of convolution sizes, but found that 32-bit performed best with a factor between 1.11× and 1.23×.

The optimizations are also summarized in Table 5.14 for clarity.

We view these optimizations as part of possible future work for increasing performance of CNN inference on the ρ-VEX platform. Other future work not directly related to inference computation will be discussed in Chapter 6.

Optimization	Expected gain	Measured gain
Hardware		
Multiply-Accumulate instruction	1.10×-1.48× on single sample computation	
SIMD Vector unit	1.52×-3.04× on single sample computation	
Multidimensional SIMD unit	Further investigation required	
Software		
Loop unrolling		1.04×-1.30× on single sample computation
Optimized BLAS routines	2.42×-3.00× on single sample computation	
Layer grouping	No loss of performance while reducing resources	1.5× reduction of resources with minimal performance loss
Switch to 32-bit arithmetic	Small benefit of 16-bit over 32-bit	32-bit outperforms 8-bit and 16-bit with 1.11 × -1.23× on single sample computation

Table 5.14: Summary of possible optimizations

6 Conclusion

In this chapter we present the conclusion of our work in this thesis. In Section 6.1, we present a summary of each chapter of the thesis. Next, in Section 6.2, the problem statement is reviewed once more and the main contributions of the thesis are presented. We also discuss if the goals that were set in Chapter 1 have been achieved. Finally, in Section 6.2.2, we conclude this chapter with possible future work based on the theme of this thesis.

6.1 Summary

Chapter 2 presents the necessary background knowledge that is required for this thesis.

In Section 2.1, we analyze the different kinds of parallelism that can be found in a computer application. The differences between Instruction Level Parallelism, Thread Level Parallelism and Data Level Parallelism are explained as well as their specific details. We subsequently introduce the ρ-VEX architecture in Section 2.2 by first discussing reconfigurable computing and VLIW architectures followed by the details of the ρ-VEX design.

Next, in Section 2.3 we briefly introduce machine learning and explain the difference between supervised and unsupervised learning as well as the details of classification problems. It is followed by an explanation of artificial neural networks in Section 2.4 and how they can be used to solve classification problems. In Section 2.5 we then explain the principles behind convolutional neural networks and their advantages over regular artificial neural networks.

Subsequently, in Section 2.6, we discuss related work that concerns the details of running CNNs on different hardware architectures, as well as common software implementations. We also discuss the steps required to efficiently run CNNs on embedded systems.

Chapter 3 develops the concept for the streaming architecture based on the ρ-VEX for running CNN inference.

In Section 3.1, convolutional neural networks are mapped to a streaming application and we list the required standard components that are necessary for the implementation. In Section 3.2 the proof of concept application is briefly described that will implement a basic CNN to demonstrate feasibility on the ρ-VEX. Next, in Section 3.3, the different ways that the ρ-VEX can take advantage of parallelism in the CNN are discussed. This can be achieved by reconfiguring the ρ-VEX based on the TLP and ILP of the CNN implementation, adding parallel streams to improve the throughput, combining layers together to reduce resource usage and unrolling loops to improve ILP of the code.

In Section 3.4 we present the design of the streaming pipeline and specific details for simulator and hardware implementations. The pipeline consists of multiple full ρ-VEX cores in series, with memories placed in between. These memories operate as FIFO buffers and are controlled by dedicated FIFO registers in the cores. The CNN parameters are also stored locally in or near the cores, the size of these memories can be accurately calculated based on the types of layers that are executed on that core.

Synchronization between contexts in a single ρ-VEX core is achieved by means of BUSY-WAIT loops, although this solution is not ideal due to the lack of atomic primitives. Synchronization between cores is also achieved with the help of BUSY-WAIT loops, but an improved wake-on-interrupt scheme is proposed. The resource requirements with regard to memory for an MNIST network are then analyzed and a four core design is settled on. The instruction and data memories are appropriately sized to execute the network on this pipeline.

Specific details relating to the simulator are then reviewed. Memory inside the simulator is one contiguous block that is divided up and assigned to individual ρ-VEX cores, each block can also be accessed by the next core by setting the most significant bit of the address.

Finally, details relating to the hardware realization are discussed. For the hardware implementation, the ρ-VEX streaming fabric is modified in such a way that it can accommodate the appropriate memory sizes for instruction and data memory. In this case only 2-issue ρ-VEX cores are used. The FIFO registers are implemented in the memory interface as to avoid modifying the processor registers.

In *Chapter 4* we then discuss the implementation of the streaming pipeline. First, in Section 4.1, we analyze the implementation of the proof of concept application on the ρ-VEX platform and discuss any difficulties that were encountered and which solutions were found. Next, in Section 4.2, the implementation of the streaming pipeline in the simulator is explained. The details of modifications to the simulator are presented, specific attention is paid to implementing multi-core support together with the memory model.

It is followed by Section 4.3 where the implementation of the code generation is presented. The reasoning behind using the Python and C programming languages is briefly discussed, after which the individual software components are presented. We also discuss the workflow of generating and running code from a high-level description, as well as an example of the syntax of such a description. In Section 4.4, we move on to hardware synthesis where first the storage requirements are determined through simulation. Finally, the necessary modifications to the streaming fabric are discussed after which a four core design is successfully synthesized. However, we conclude that more work is required to complete the implementation.

Chapter 5 presents the final evaluation of the work. We begin in Section 5.1 with a verification of the correctness of the implementation. It was achieved by comparing the network output with a reference output from a Caffe distribution on a desktop machine, which were found to be equal. We then move on to the performance measurements in Section 5.2. We measure the throughput and latency of the MNIST network with the

help of the simulator, as well as the ILP present in the code. We also analyze the benefits from pipelining the cores as a stream.

It is followed by possible optimizations to the architecture in Section 5.3. We discuss both hardware and software optimizations. First, we discuss how a dedicated multiply-add operation could be implemented in the ρ-VEX to improve the performance of common CNN operations. Subsequently, we analyze how a SIMD vector unit could improve performance by operating on multiple data per cycle. Finally, we look at a multidimensional SIMD unit with polymorphic registers as an interesting option for accelerating convolution type operations.

On the software side, we first discuss how loop unrolling could improve performance, followed by a brief review of how dedicated ρ-VEX BLAS routines could impact performance based on results from literature. Next, we investigate how grouping CNN layers together on cores could reduce resource usage while not impacting performance significantly. Finally, we analyze if lower precision calculations yield any performance benefits on the ρ-VEX platform.

6.2 Main contributions

In this section we will review the main contributions of the thesis and investigate how many of our original goals have been met. First, we revisit the problem statement introduced in Chapter 1:

How can the ρ-VEX processor be utilized to perform CNN inference and how can performance be improved?

We demonstrated that the ρ-VEX processor can perform CNN inference on a simple network using our proof-of-concept implementation. It was achieved by quantizing the parameters to fixed point representation in such a way that the network can be executed on the ρ-VEX platform.

To improve performance we proposed creating a streaming pipeline of full 8-issue ρ-VEX cores. In this way, each ρ-VEX core can run one or more CNN operations and reconfigure itself based on the present DLP, TLP or ILP in the CNN code. The pipelining then serves as a way to increase the throughput of the network by running multiple operations in parallel on ρ-VEX cores and buffering intermediate data in between. This design was implemented in a simulator together with a code generator that generates all necessary code based on the high-level description of a CNN.

To build this in hardware, we took advantage of the existing ρ-VEX streaming fabric to implement our pipeline. Using a four core stream, the pipeline was implemented by appropriately sizing the instruction and data memories. It also required modification of the debug bus to accommodate the increased memory sizes. Because the fabric only supports single 2-issue ρ-VEX cores, the reconfiguration features cannot be used. Finally, the design was successfully synthesized for a Virtex-6 FPGA [67] at 200 MHz.

6.2.1 Contributions

The main contributions of this thesis are listed here:

- Created a proof of concept implementation of CNN inference on the ρ-VEX platform, tested both in simulator and hardware. It was verified to match the reference on the entire MNIST test dataset.
- Added multi-core support, streaming memories and configuration file loading to the ρ-VEX simulator. The code of the simulator was reorganized in such a way that all processor state is contained in a single data structure. The number of cores and its configuration parameters can now be passed through a configuration file.
- Designed a streaming pipeline for CNN inference based around reconfigurable 8-issue ρ-VEX cores and tested this in a simulator.
- Created a code generation tool that can generate the required code for CNN inference on the streaming pipeline based on a high-level description file. A subset of CNN layers was implemented as building blocks to facilitate this.
- Adapted the ρ-VEX streaming fabric hardware description to enable execution of our CNN inference code by increasing size of instruction and data memories and adapting the debug bus.
- Synthesized a four core 2-issue ρ-VEX stream with memories sized for CNN inference of the MNIST dataset at a clock speed of 200 MHz.
- Investigated different improvements that can be made to increase the performance of CNN inference on the ρ-VEX platform.

6.2.2 Future work

In this section we will review possible areas of future work based on the work done in this thesis. We view the optimizations in Chapter 5 also part of the possible future work, but we will not reiterate all of them here and instead refer to that chapter.

- The streaming fabric was only synthesized but not tested in hardware due to time constraints, it would greatly increase the potential of this project if hardware functionality is verified. However, further development would be required, as the FIFO registers were not yet implemented.
- Currently, the parameters in the code generation are loaded as RAM images, however this requires arbitrary spacing in the instruction memory, which is wasted space. The code generation should be changed in such a way that the input parameters are compiled directly into the binary to simplify this process.
- Implementing support for atomic synchronization primitives in the ρ-VEX core would improve thread synchronization capabilities between contexts and aid in preventing deadlocks and race conditions. Currently, a BUSY-WAIT loop must be

used that is not thread-safe. Atomic primitives can be implemented by adding a load-link/store-conditional or compare-and-swap instruction to the ρ-VEX instruction set that executes atomically.

- A hardware floating point unit in the ρ-VEX processor would remove the need for quantizing the parameters and enable it to run a CNN directly. It would however lead to an increase in the size of the core and this is not desirable in certain conditions. A parametrizable floating point unit would therefore be an interesting option, making it configurable per lane and even switching on and off support for certain floating point operations in a straightforward way. As a CNN mostly relies on multiply-add operations, some instructions such as division may not have to be implemented in hardware and software support could suffice.

- Multiple pipelines in parallel could be used to improve the latency of a single inference pass by distributing the workload of each pipeline stage over multiple cores. By creating a parametrizable design that can both scale the pipeline length and the number of parallel pipelines, we can both improve throughput and latency based on the available resources. However, it raises questions on how the workload should be divided and how much communication between pipelines is necessary, which should be investigated first.

- Because local memories do not scale for larger networks, it would be necessary to stream in parameter data from external memory when required, as well as buffering intermediate data. It would require the development of a memory bus that connects to the cores to stream in parameter data and stream out buffer data. To reduce the impact of the latency of external memory, the timing of the transfers of the data would need to be carefully managed.

- A more simple design based around a PE array such as seen in [22] could be implemented using 2-issue ρ-VEX cores or a single 8-issue reconfigurable ρ-VEX core. This is more in line with the design of most CNN research accelerators, as there are usually not enough available resources on FPGA or ASIC devices to implement large arrays of processors. However, it would also mean implementing an efficient local buffering hierarchy and external memory interface.

- As proposed in Chapter 5, SIMD vector extensions would be a useful addition to the ρ-VEX platform. Because we want to operate on multiple data elements in parallel each cycle, it will place demands on the memory interface, therefore any development of vector extensions would also require the implementation of a memory interface with throughput optimizations such as burst reads.

- Because of the nature of the applications of CNNs, one could imagine these would need to be combined with other operations such as image preprocessing or output rendering. It would be especially useful in an embedded system, as we want to make use of the available resources as efficiently as possible. As such, the streaming pipeline of single full ρ-VEX cores could be divided up into multiple (1 to 4) separate streams by running different code on each context. For example, an input

image can be filtered on the first stream after which it is run through the CNN on the second stream and finally the third stream can be used to render the result to a display. These streams can also be sized based on their workload by assigning more or less contexts to them. It extends the idea of the single core ρ-VEX where contexts are appropriately sized based on their workload, but now applies to an entire stream.

Bibliography

[1] Delft University of Technology, Computer Engineering Department, "ρ-VEX: the dynamically reconfigurable VLIW processor," http://rvex.ewi.tudelft.nl/, accessed: 2018-02-16.

[2] F.-F. Li, A. Karpathy, and J. Johnson, "CS231n convolutional neural networks for visual recognition," http://cs231n.github.io/neural-networks-1/, Course Notes, Accessed: 2018-03-07.

[3] R. D. Reed and R. J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[5] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning – ICANN 2014*. Cham: Springer International Publishing, 2014, pp. 281–290.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[8] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[9] J. Hoozemans, J. Johansen, J. V. Straten, A. Brandon, and S. Wong, "Multiple contexts in a multi-ported VLIW register file implementation," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–6.

[10] O. Russakovsky, J. Deng *et al.*, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[11] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov 2012.

[12] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at Facebook: A datacenter infrastructure perspective," *Unpublished*, 2018, available at https://research.fb.com/publications/applied-machine-learning-at-facebook-a-datacenter-infrastructure-perspective/.

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[14] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[16] Nvidia Corporation, "NVIDIA Tesla P100 architecture," Whitepaper https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016, accessed: 2018-02-20.

[17] Intel Corporation, "Intel architecture instruction set extensions and future features programming reference," Technical document https://software.intel.com/en-us/isa-extensions, Jan 2018, accessed: 2018-02-20.

[18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 WORKSHOPS*, June 2011, pp. 109–116.

[19] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[20] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[21] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303

[22] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung, "Accelerating deep convolutional neural networks using specialized hardware," https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/, February 2015.

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[24] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Institute of Electrical and Electronics Engineers (IEEE), May 2016, pp. 40–47. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2016.22

[25] Xilinx Corporation, "MicroBlaze soft processor core," https://www.xilinx.com/products/design-tools/microblaze.html, accessed: 2018-02-20.

[26] S. Wong, T. van As, and G. Brown, "ρ-VEX: A reconfigurable and extensible softcore VLIW processor," in *Proc. International Conference on Field-Programmable Technology*, Taipei, Taiwan, December 2008.

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

[28] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[29] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," http://yann.lecun.com/exdb/mnist/, accessed: 2018-02-20.

[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[31] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[32] Hewlett Packard, "VEX toolchain," http://www.hpl.hp.com/downloads/vex/, accessed: 2018-02-16.

[33] J. van Straten, "A dynamically reconfigurable VLIW processor and cache design with precise trap and debug support," Master's thesis, Delft University of Technology, Delft, The Netherlands, May 2016.

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

[35] J. Giraldo, A. L. Sartor, L. Carro, S. Wong, and A. Beck, "Evaluation of energy savings on a VLIW processor through dynamic issue-width adaptation," in *Proc. 2015 International Symposium on Rapid System Prototyping*, Pittsburgh, PA, USA, October 2015, pp. 11–17.

[36] A. Brandon and S. Wong, "Support for dynamic issue width in VLIW processors using generic binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, France, March 2013, pp. 827 – 832.

[37] Newlib, "The Newlib homepage," https://sourceware.org/newlib/, accessed: 2018-02-16.

[38] Cobham Gaisler AB, "GRLIB SoC library," http://www.gaisler.com/index.php/products/ipcores/soclibrary, accessed: 2018-02-16.

[39] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[40] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, Dec 1989. [Online]. Available: https://doi.org/10.1007/BF02551274

[41] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, jan 1962. [Online]. Available: https://doi.org/10.1113/jphysiol.1962.sp006837

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[43] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, Nov 2017.

[44] M. Thoma, "Analysis and optimization of convolutional neural network architectures," Master's thesis, Karlsruhe Institute of Technology, jul 2017, arXiv:1707.09725 [cs.CV] Available: http://arxiv.org/abs/1707.09725.

[45] Intel Corporation, "Intel math kernel library," https://software.intel.com/en-us/mkl, accessed: 2018-02-19.

[46] Nvidia Corporation, "cuBLAS — NVIDIA developer," https://developer.nvidia.com/cublas, accessed: 2018-02-19.

[47] ——, "NVIDIA cuDNN — NVIDIA developer," https://developer.nvidia.com/cudnn, accessed: 2018-02-19.

[48] Intel Corporation, "Intel math kernel library for deep neural networks," https://01.org/mkl-dnn, accessed: 2018-02-19.

[49] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 247–257. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[50] S. W. Park, J. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. J. Yoo, "An energy-efficient and scalable deep learning/inference processor with tetra-parallel MIMD architecture for big data applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, no. 6, pp. 838–848, Dec 2015.

[51] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[52] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[53] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[54] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, may 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[55] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[56] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[57] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[58] S. I. Venieris and C. S. Bouganis, "Latency-Driven Design for FPGA-based Convolutional Neural Networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–8.

[59] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 13–19.

[60] J. Kurzak, W. Alvaro, and J. Dongarra, "Optimizing matrix multiplication for a short-vector SIMD architecture CELL processor," *Parallel Computing*,

vol. 35, no. 3, pp. 138 – 150, 2009, revolutionary Technologies for Acceleration of Emerging Petascale Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016781910900012X

[61] *ML605 Hardware User Guide*, Ug534 (v1.8) ed., Xilinx Corporation, oct 2012, available at: https://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf Accessed: 2018-02-26.

[62] IEEE, "IEEE standard for information technology- POSIX r ADA language interfaces- part 1: Binding for system application program interface (API)- amendment 1: Realtime extensions," *IEEE Std 1003.5b-1996 (Includes IEEE Std 1003.5-1992)*, pp. 1–529, 1997.

[63] Python Software Foundation, "Welcome to python.org," https://www.python.org/, accessed: 2018-02-23.

[64] P. McGuire, "pyparsing," https://pythonhosted.org/pyparsing/, accessed: 2018-02-23.

[65] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice Hall Press, 1988.

[66] B. Parhami, *Algorithms and Design Methods for Digital Computer Arithmetic*, 2nd ed. New York: Oxford University Press, 2012.

[67] *Virtex-6 Family Overview*, Ds150 (v2.5) ed., Xilinx Corporation, aug 2015, available at: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf Accessed: 2018-02-26.

[68] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: a vector extension to the alpha architecture," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 281–292.

[69] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, Mar 2014.

[70] C. Ciobanu, G. Kuzmanov, G. Gaydadjiev, and A. Ramirez, "A polymorphic register file for matrix operations," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2010, pp. 241–249.

[71] C. B. Ciobanu and G. N. Gaydadjiev, "Separable 2D convolution with polymorphic register files," in *Architecture of Computing Systems – ARCS 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 317–328.

[72] OpenBLAS, "OpenBLAS: An optimized BLAS library," http://www.openblas.net/, accessed: 2018-02-28.

[73] gemmlowp, "Low-precision matrix multiplication," https://opensource.google.com/projects/gemmlowp, accessed: 2018-02-28.

[74] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. La Baule (France): Université de Rennes 1, Oct. 2006. [Online]. Available: https://hal.inria.fr/inria-00112631

[75] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[76] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[77] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

Synthesis report streaming pipeline

A

This appendix contains the synthesis report of the four core 2-issue pipeline.

ml605_streaming_platform Project Status (02/05/2018 - 13:25:51)			
Project File:	stream.xise	**Parser Errors:**	No Errors
Module Name:	ml605_streaming_platform	**Implementation State:**	Programming File Generated
Target Device:	xc6vlx240t-1ff1156	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	6780 Warnings (29 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	12,359	301,440	4%	
Number used as Flip Flops	12,358			
Number used as Latches	1			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	32,468	150,720	21%	
Number used as logic	31,812	150,720	21%	
Number using O6 output only	28,611			
Number using O5 output only	677			
Number using O5 and O6	2,524			
Number used as ROM	0			
Number used as Memory	408	58,400	1%	
Number used as Dual Port RAM	24			
Number using O6 output only	24			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used as Single Port RAM	0			
Number used as Shift Register	384			
Number using O6 output only	384			
Number using O5 output only	0			
Number using O5 and O6	0			

Number used exclusively as route-thrus	248			
Number with same-slice register load	52			
Number with same-slice carry load	196			
Number with other load	0			
Number of occupied Slices	12,057	37,680	31%	
Number of LUT Flip Flop pairs used	33,773			
Number with an unused Flip Flop	22,058	33,773	65%	
Number with an unused LUT	1,305	33,773	3%	
Number of fully used LUT-FF pairs	10,410	33,773	30%	
Number of unique control sets	161			
Number of slice register sites lost to control set restrictions	729	301,440	1%	
Number of bonded IOBs	13	600	2%	
Number of LOCed IOBs	12	13	92%	
Number of RAMB36E1/FIFO36E1s	357	416	85%	
Number using RAMB36E1 only	357			
Number using FIFO36E1 only	0			
Number of RAMB18E1/FIFO18E1s	60	832	7%	
Number using RAMB18E1 only	60			
Number using FIFO18E1 only	0			
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number used as BUFGCTRLs	0			
Number of ILOGICE1/ISERDESE1s	0	720	0%	
Number of OLOGICE1/OSERDESE1s	0	720	0%	
Number of BSCANs	0	4	0%	
Number of BUFHCEs	0	144	0%	
Number of BUFIODQSs	0	72	0%	
Number of BUFRs	0	36	0%	
Number of CAPTUREs	0	1	0%	
Number of DSP48E1s	16	768	2%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE1s	0	20	0%	
Number of IBUFDS_GTXE1s	0	12	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	0	18	0%	

Number of IODELAYE1s	0	720	0%	
Number of MMCM_ADVs	1	12	8%	
Number of PCIE_2_0s	0	2	0%	
Number of STARTUPs	1	1	100%	
Number of SYSMONs	0	1	0%	
Number of TEMAC_SINGLEs	0	4	0%	
Average Fanout of Non-Clock Nets	3.80			

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Mon Feb 5 12:08:09 2018	0	6766 Warnings (29 new)	1570 Infos (2 new)
Translation Report	Current	Mon Feb 5 12:11:35 2018	0	0	4 Infos (0 new)
Map Report	Current	Mon Feb 5 12:39:18 2018	0	4 Warnings (0 new)	11 Infos (0 new)
Place and Route Report	Current	Mon Feb 5 12:54:35 2018	0	6 Warnings (0 new)	1 Info (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Mon Feb 5 13:06:44 2018	0	0	4 Infos (0 new)
Bitgen Report	Current	Mon Feb 5 13:25:49 2018	0	4 Warnings (0 new)	1 Info (0 new)

Secondary Reports		[-]
Report Name	**Status**	**Generated**
WebTalk Log File	Current	Mon Feb 5 13:25:50 2018

Date Generated: 03/02/2018 - 13:54:45

Synthesis report standalone 8-issue ρ-VEX

B

This appendix contains the module level utilization of the synthesized 8-issue ρ-VEX core for the ML605 development board. The utilization of the multiplier unit is highlighted in Figure B.1.

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48E1	BUFG	BUFIO	BUFR
ml605		35/18050	106/18123	111/46706	0/1197	0/328	0/16	1/16	0/0	0/0
rvex_standalone.dbg_bus_demux		21/21	2/2	37/37	0/0	0/0	0/0	0/0	0/0	0/0
rvex_standalone.rvex_inst		2/17826	0/17694	2/46133	0/1157	0/328	0/16	0/15	0/0	0/0
core_gen.core		302/15...	667/17558	430/40743	0/1157	0/134	0/16	0/15	0/0	0/0
core		16/15685	0/16891	21/40313	0/1157	0/134	0/16	0/15	0/0	0/0
cfg_inst		85/108	64/114	120/161	0/0	0/0	0/0	0/0	0/0	0/0
creg_inst		28/278	0/146	30/520	0/0	0/0	0/0	0/0	0/0	0/0
cxreg_inst		2553/2...	4164/4164	6673/6673	0/0	0/0	0/0	0/0	0/0	0/0
gbreg_inst		291/291	250/250	743/743	0/0	0/0	0/0	0/0	0/0	0/0
gpreg_inst		281/5752	516/1390	284/14298	0/0	0/128	0/0	0/0	0/0	0/0
ibuf_gen.ibuf_inst		84/84	203/203	344/344	0/0	0/0	0/0	0/0	0/0	0/0
pls_inst		162/6104	12/9043	437/16505	0/1157	2/6	0/16	15/15	0/0	0/0
cxplif_inst		1220/1...	0/0	2628/3034	0/0	0/0	0/0	0/0	0/0	0/0
dmsw_gen[0]....		15/15	1/1	33/33	0/0	0/0	0/0	0/0	0/0	0/0
dmsw_gen[1]....		19/19	1/1	38/38	0/0	0/0	0/0	0/0	0/0	0/0
dmsw_gen[2]....		12/12	1/1	34/34	0/0	0/0	0/0	0/0	0/0	0/0
dmsw_gen[3]....		23/23	31/31	40/40	0/0	0/0	0/0	0/0	0/0	0/0
limm_inst		8/8	0/0	8/8	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[0].pl_inst		211/520	706/922	577/1347	89/89	0/0	0/2	0/0	0/0	0/0
alu_inst		253/253	178/178	578/578	0/0	0/0	0/0	0/0	0/0	0/0
brku_gen....		25/25	0/0	129/129	0/0	0/0	0/0	0/0	0/0	0/0
memu_ge...		4/4	5/5	19/19	0/0	0/0	0/0	0/0	0/0	0/0
mulu_gen....		27/27	33/33	44/44	0/0	0/0	2/2	0/0	0/0	0/0
pl_gen[0].pl_in...		2/2	0/0	4/4	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[0].pl_in...		0/0	0/0	8/8	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[0].pl_in...		1/1	0/0	11/11	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[1].pl_inst		287/667	1029/1317	726/1629	193/193	0/0	0/2	0/0	0/0	0/0
pl_gen[1].pl_in...		1/1	0/0	3/3	0/0	1/1	0/0	0/0	0/0	0/0
pl_gen[1].pl_in...		3/3	0/0	14/14	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[2].pl_inst		201/454	712/928	607/1377	89/89	0/0	0/2	0/0	0/0	0/0
pl_gen[2].pl_in...		1/1	0/0	3/3	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[2].pl_in...		0/0	0/0	8/8	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[2].pl_in...		1/1	0/0	11/11	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[3].pl_inst		277/666	1036/1324	789/1703	202/202	0/0	0/2	0/0	0/0	0/0
pl_gen[3].pl_in...		1/1	0/0	3/3	0/0	1/1	0/0	0/0	0/0	0/0
pl_gen[3].pl_in...		1/1	0/0	14/14	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[4].pl_inst		195/462	712/928	591/1361	89/89	0/0	0/2	0/0	0/0	0/0
pl_gen[4].pl_in...		0/0	0/0	3/3	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[4].pl_in...		1/1	0/0	8/8	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[4].pl_in...		1/1	0/0	11/11	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[5].pl_inst		288/637	1036/1324	791/1702	202/202	0/0	0/2	0/0	0/0	0/0
pl_gen[5].pl_in...		1/1	0/0	3/3	0/0	1/1	0/0	0/0	0/0	0/0
pl_gen[5].pl_in...		4/4	0/0	14/14	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[6].pl_inst		215/495	714/930	600/1371	89/89	0/0	0/2	0/0	0/0	0/0
pl_gen[6].pl_in...		1/1	0/0	3/3	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[6].pl_in...		0/0	0/0	8/8	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[6].pl_in...		4/4	0/0	11/11	0/0	0/0	0/0	0/0	0/0	0/0
pl_gen[7].pl_inst		258/523	1036/1324	703/1726	204/204	0/0	0/2	0/0	0/0	0/0
pl_gen[7].pl_in...		1/1	0/0	3/3	0/0	1/1	0/0	0/0	0/0	0/0
pl_gen[7].pl_in...		1/1	0/0	13/13	0/0	0/0	0/0	0/0	0/0	0/0
sbit_inst		17/17	0/0	91/91	0/0	0/0	0/0	0/0	0/0	0/0
trap_routing.tr...		73/73	0/0	418/418	0/0	0/0	0/0	0/0	0/0	0/0
trace_gen.trace_inst		499/499	1581/1581	1048/1048	0/0	0/0	0/0	0/0	0/0	0/0
debug_bus_demux_gen_sa...		31/31	5/5	151/151	0/0	0/0	0/0	0/0	0/0	0/0
dmem_block.dmem_arbite...		9/9	5/5	54/54	0/0	0/0	0/0	0/0	0/0	0/0
dmem_block.dmem_arbite...		13/13	3/3	27/27	0/0	0/0	0/0	0/0	0/0	0/0
dmem_block.dmem_ram		622/622	14/14	1739/1739	0/0	64/64	0/0	0/0	0/0	0/0

Design Summary

Figure B.1: Module utilization of multiplier in 8-issue ρ-VEX synthesis.