NTNU – Trondheim
Norwegian University of
Science and Technology

Hardware Acceleration of Convolutional Neural Networks

Magnus Halvorsen

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Donn Morrison, IDI
Co-supervisor: Yaman Umuroglu, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Convolutional neural networks have been widely employed for image recognition applications because of their high accuracy, which they achieve by emulating how our own brain recognizes objects. The possibility of making our electronic devices recognize their surroundings have spawned a vast number potential of useful applications, including video surveillance, mobile robot vision, image search in data centres, and more. The increasing usage of such applications in mobile platforms and data centres have led to a higher demands for methods that can compute these computational-insensitive networks in a fast and power efficient way. One such method is by using application specific hardware accelerators.

In this report we will present such an accelerator, and use it to compute a neural network that can recognize hand-written digits.

Sammendrag

Konvolusjonsnettverk er en veletablert metode som brukes i objektidentifisering av bilder. Dette gjr den ved emulere hvordan hjernen vr bruker et nettverk av neuroner til gjennkjenne objekter med synet vrt. Denne muligheten til f de elektroniske enhetene vre til gjennkjenne omgivelsene sine er gitt en kraftig kning i applikasjoner som utnytter dette. Dette inkluderer videoovervkning, robotsyn, bildesk i datavarehus, og mer. Denne kningen i mulige bruksomrdet innenfor mobile platformer og datavarehus har frt til et kt behov for raske og energieffektive metoder for prossesere slike nettverk. En slik metode er lage applikasjonsspesifikke maksinvareakseleratorere.

I denne rapporten vil vi presentere en slik akselerator, og bruke den til prossessere et nettverk som gjennkjenner hndskrevne tall.

Assignment

Candidate name: Magnus Halvorsen

Assignment title: Hardware Acceleration of Convolutional Neural Networks

Supervisors: Donn Alexander Morrison and Yaman Umuroglu

Assignment text:

This project will explore the design and implementation of convolutional neural networks (CNNs) in hardware with the intention of improving energy efficiency over traditional implementation in software on a general-purpose CPU. The overall goal is to build a standalone system that (time permitting) could be trained interactively by a user (e.g., to recognise handwriting or faces from a webcam video stream) and then demonstrate learned patterns through recognition of unseen samples.

The energy efficiency of the hardware implementation should be evaluated against software equivalent running on a general-purpose CPU and this evaluation should constitute a major aspect of the report.

The suggested platform is the Zynq FPGA board, but the student can also investigate and weigh the advantages and disadvantages other platforms such as SHMAC.

Contents

1 Introduction **2**
- 1.1 Motivation . . . 2
- 1.2 Assignment Interpretation . . . 3
- 1.3 Report structure . . . 4

2 Background **5**
- 2.1 Artificial Neural Networks . . . 5
  - 2.1.1 Definition . . . 6
  - 2.1.2 Training . . . 7
  - 2.1.3 Issues with object recognition . . . 10
- 2.2 Convolutional Neural Network . . . 10
  - 2.2.1 Definition . . . 10
  - 2.2.2 Training . . . 13
- 2.3 Potential for parallelism . . . 15
- 2.4 ZedBoard . . . 16

3 Related Work **18**
- 3.1 Convolutional Neural Networks . . . 18
- 3.2 Convolutional Neural Network in Hardware . . . 19

4 Architecture **22**
- 4.1 Network Topology and Dataset . . . 23
- 4.2 What to accelerate . . . 24
- 4.3 Software Architecture . . . 25
  - 4.3.1 Network Software . . . 25
  - 4.3.2 Hardware driver . . . 26
- 4.4 Hardware Architecture . . . 32
  - 4.4.1 Overview of Hardware Architecture . . . 33
  - 4.4.2 Accelerator Interface . . . 34
  - 4.4.3 The Accelerator's Processing Unit . . . 35

4.4.4 The Convoluter 37
4.4.5 The Hyperbolic Tangent 38
4.4.6 The Average Pooler 39

5 Results and Discussion **42**
5.1 Hardware Resources 42
5.2 Performance 43
5.2.1 Setup 43
5.2.2 Discussion 44

6 Future work **49**

7 Conclusion **52**

List of Figures

2.1 Neuron 7
2.2 An Artificial Neural Network. 7
2.3 The LeNet-5 11
2.4 Convolution and subsampling/max-pooling operation 13
2.5 Convolutional layer operation 14
2.6 Zyng-7000 system architecture. 17

4.1 The topology of the implemented network. 22
4.2 Feature map dependencies 24
4.3 Software architecture 26
4.4 FIFO order 27
4.5 Tx Bd ring 28
4.6 The system architecture 32
4.7 The interface of the accelerator. 34
4.8 Controller's states 34
4.9 Accelerator's processing unit 36
4.10 The Convoluter 38
4.11 Convolution example 39
4.12 The Average Pooler 41

5.1 Accelerator performance 45
5.2 Accelerator performance, layer 1 and 2 46
5.3 Accelerator performance, with DMA fix 46
5.4 Performance comparison 47
5.5 Performance when only processing layer 1 and 2. 48

List of Tables

4.1 Number of connections in the network 25
4.2 Controller table 35
4.3 The constant used for the hyperbolic tangent approximation. . . . 40
4.4 The piecewise linear approximation of the hyperbolic tangent. . . . 40

5.1 Resource usage 43
5.2 Available resources 43

Chapter 1

Introduction

1.1 Motivation

A *Convolutional Neural Network* (CNN) is a deep-learning algorithm architecture that has become increasingly popular in the last decade. It is considered a state of art technique for object recognition in images and sound, and it is applied in application such as video surveillance, mobile robot vision, image search in data centers, and more [1] [2] [3] [4]. With the Internet-of-Things and today's tremendous amount of devices able to capture pictures and videos, the potential for CNNs have vastly increased. By making our devices able to recognize its surroundings, there could be a numerous amount of potential interesting applications.

Albeit CNNs perform great in terms of accuracy (see Chapter 3), they are very computational heavy, which have limited their usability until recent years. The computational structure of neural networks is highly parallelizable, which, when exploited, can greatly increase performance. It is for this reason that *general-purpose central processing units* (GPCPUs) performs poorly when computing such networks, as they are primarily designed for effective serial computations, and are thus unable to exploit its parallel structure. *Field-programmable gate arrays* (FPGAs), *graphic processor units* (GPUs) and *application-specific integrated circuits* (ASICs) are hardware components that are (or can be) built to heavily exploit parallelism, and have been shown to greatly outperform CPUs in parallel applications [5].

While GPUs performs incredibly well on parallel applications, they have a major drawback: power consumption. With power being the primary financial cost of data centers and mobile devices having to operate on a limited power budget [6], GPUs can be unsuitable for several applications. Thus, for CNN

applications that require lower power and high performance, FPGA and ASIC accelerators have become increasingly popular (see Section 3.2). Simplistically, an ASIC is an processor that is made specifically for one application, while a FPGA is a component that contains a set of programmable logic blocks that can be configured to have the same behavior as any arbitrary circuit. I.e. a FPGA is a reconfigurable ASIC.

In our previous work [7], we investigated the mathematical model behind neural networks and purposed a unimplemented accelerator architecture. In this project, we have implemented a tweaked version of this architecture on a Zynq FPGA, and constructed all the supporting hardware and software components needed to get it running. In order to test its capabilities, we have used it to compute the LeNet-5 [8], which recognizes handwritten digits. Our system has shown to be 5.6x times as fast and power efficient than an ARM Cortex-A9 processor, and 5x as power efficient as an Intel Core i7 4710HQ CPU on certain parts of the processing.

1.2 Assignment Interpretation

Based on the assignment description text, the following main tasks were identified:

Task 1 ***(mandatory)*** Implement a hardware accelerator for a Convolutional Neural Network, with the intention of improving energy efficiency.

Task 2 ***(mandatory)*** Compare our accelerator to an equivalent pure-software implementation on a general-purpose CPU, primarily in terms of power consumption.

Task 3 ***(optional)*** Implement said system on a Zynq FPGA board, but weigh the advantages and disadvantages of other platforms, such as SHMAC or other FPGA platforms.

Task 4 ***(optional)*** Extend the system to be able to recognize objects from a web-cam stream.

1.3 Report structure

For the convenience of the reader, we will here provide a quick overview of the topic of the report's chapters.

Chapter 2: Background gives an introduction to the mathematical model of Artificial Neural Networks and Convolutional Neural Networks.

Chapter 3: Related Work gives an overview of the state of the art CNNs and the most relevant recent hardware implementations.

Chapter 4: Architecture presents our implemented design for a CNN hardware accelerator.

Chapter 5: Results and Discussion compares our design with an equivalent implementation on a ARM and laptop CPU, in terms of performance and energy efficiency, and the hardware resource usage of our design. The chapter will also provide our analysis of the results.

Chapter 6: Future Work presents how our design can be further improved.

Chapter 7: Conclusion provides concluding remarks and a summary of the identified tasks.

Chapter 2

Background

In order to be able to accelerate a system, it is important to understand the mathematical model behind it. For this reason, we have included Section 2.1 and 2.2 from our previous work [7], which purposed a theoretical accelerator architecture. The sections will introduce the fundamental mathematics and concepts behind the *Convolutional Neural Network* (CNN) model. They give a basic introduction to both general neural networks and *CNNs*. Section 2.4 will present the hardware platform we used to implement our accelerator.

2.1 Artificial Neural Networks

An *Artificial Neural Network* (ANN) [9][10] is a computational model that is used for machine learning and pattern recognition. The name and basic concept is inspired by how the animal brain uses a network of neurons to recognize and classify objects. Depending on the input, different neurons *activate* (or *fire*), making the brain able to decide what kind of pattern it is detecting.

An ANN can intuitively be viewed as a probabilistic classifier. Depending on the input data, it will calculate the probability that the data belongs to a certain *class* (e.g. an object in an image or an investment decision). The network can be trained to recognize different classes by being provided a set of labeled training data, e.g. a set of faces and a set of non-faces. It can then learn to decide whether an image contains a face or not. This is called supervised learning. The network can also be trained unsupervised, by providing it with a set of unlabeled images. The latter technique is used to find hidden structures in the data, by learning the network to recreate the input. But for this project only supervised learning is relevant.

2.1.1 Definition

An ANN consists of a number of layers containing a set of so-called *neurons* (see Figure 2.1), also known as *units*. A neuron takes in a set of values as input (e.g. image pixels), where each value is associated with a respective weight. The input and the weights are multiplied and summed, and the result is used to calculate a non-linear *activation function*. Formally a neuron's input and output is defined as:

$$Input : \{x_1, x_2, \ldots, x_n\} = \mathbf{x} \tag{2.1}$$

$$Output : f(\mathbf{w^T x}) = f(\sum_{i=1}^{n} w_i x_i + b) = o \tag{2.2}$$

Where **w** is the vector containing connection weights and b is the neuron bias. *f(...)* is the activation function, which eumulates the activation of a neuron in the brain, i.e. it decides whether the neuron is on or off. It also causes the values in the network to have a reasonable value interval. *f(...)* tends to be either:

$$Sigmoid : f(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

$$\text{Hyperbolic tagent:} f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.4}$$

The reason these functions are used is that they have the non-linear property, which increases the expressivness of the network. Thus reducing the number of neurons the network needs to solve a given problem. In addition both function have ranges [0, 1] and [-1, 1], respectively, which translates well into probability computation. I.e. you can view the value of the activation function as the probability of that neuron activating.

An ANN consist of n_l layers, each containing a set of neurons. The first layer is *the input layer*, and the last layer is *the output layer*. The layers in between are called *the hidden layers*. Each layer uses the previous layer's output as input. The input layer is provided with the initial input and uses it to calculate the activation function for each of its neurons. The result is propagated to the first hidden layer, and continues up until it reaches the output layer, which provides the final output. This is known as a *feedforward neural network.*

The network takes in two parameters:

$$(\mathbf{W}, \mathbf{b}) = (\mathbf{W}^{(1)}, b^{(1)}, \mathbf{W}^{(2)}, b^{(2)}, \ldots, \mathbf{W}^{(n_l)}, b^{(n_l)}) \tag{2.5}$$

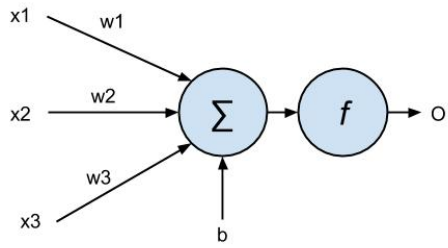

Figure 2.1: A single neuron with three inputs.

Where $\mathbf{W}$ is a 3-dimensional matrix containing the weight matrix for each layer. $\mathbf{W}^{(l)}$ contains the weight matrix for the weights going from layer l to $l+1$. E.g. in the case of Figure 2.2, $\mathbf{W}^{(1)} \in \Re^{3\times4}$ and $\mathbf{W}^{(2)} \in \Re^{4\times2}$.

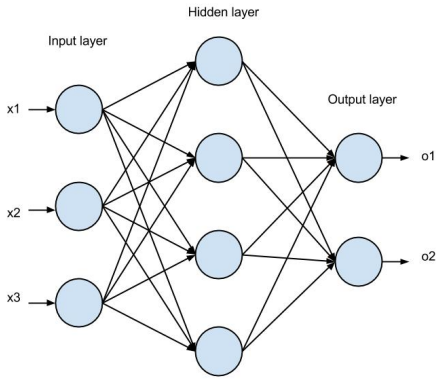

Figure 2.2: An Artificial Neural Network.

2.1.2 Training

During the training of the network, it is the parameters $(\mathbf{W}, \mathbf{b})$ that are altered in order to adapt the network to the training data. This is done by providing the network with a set of training samples, where an input and an expected output is provided. By using a cost function we can then figure out how we should tune our weights and biases in order to reduce the error rate. In other words, our goal is to minimize a cost function over a set of training samples. This can be done by using *gradient descent* and the *backpropagation algorithm* [11][12][8].

Let the cost function for a single training example (x, y) be defined as:

$$Cost(\mathbf{W}, \mathbf{b}; x, y) = \frac{1}{2}(h_{\mathbf{W},\mathbf{b}}(x) - y)^2 \tag{2.6}$$

Where x is the input, $h_{\mathbf{w},b}(x)$ is the actual output of our network and y is the correct output. Then the cost function for m training examples $((x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m))$ is:

$$Cost(\mathbf{W}, \mathbf{b}) = \frac{1}{m}\sum_{i=1}^{m} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\mathbf{w}_{ji}^l)^2 \tag{2.7}$$

Where the first term is simply the average sum-of-squares error. The second term is the *regularization term*, or *weight decay term*, which tends to reduce *overfitting*. ANNs have a vast number of parameters, i.e. weights, which makes it susceptible to random noise. This can greatly reduce the networks ability to provide correct predictions, but this can be mended by the regularization term.

Based on this we can use gradient descent to compute how we should alter the weights in order to reduce the cost function. One iteration of gradient descent updates $\mathbf{w}$ and b as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha\frac{\partial}{\partial w_{ij}^{(l)}}Cost(\mathbf{W}, \mathbf{b}) \tag{2.8}$$

$$b^{(l)} = b^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}Cost(\mathbf{W}, \mathbf{b}) \tag{2.9}$$

Where α is the learning rate, which is a predetermined constant. w_{ij}^l denotes the weight between neuron j in layer l, and neuron i in layer l+1. b_i^l denotes the bias associated with neuron i in layer l+1.

Note that this would only make us able to compute the gradient for the output layer. In order to perform gradient descent on the hidden layers, we need to propagate the error from the output layer backwards, to the hidden layers. For this we use the *backpropagation algorithm*. Let $o_i^{(l)}$ denote the output of the ith neuron in layer l, and $z_k^{(l)}$ is the weighted sum of the inputs plus the bias for the kth neuron in layer l. Then the *backpropagation algorithm* can be formalized as follows:

1. Perform a feedforward pass, computing the output of every layer.
2. For each output neuron k in the output layer, compute *the error term*:

$$\delta_k = \frac{\partial}{\partial z_k^{(n_l)}}Cost(\mathbf{W}, \mathbf{b}; x, y) = -o_k^{n_l}(1 - o_k^{n_l})(y_k - o_k^{n_l}) \tag{2.10}$$

3. For each hidden layer $l = n_l - 1, n_l - 2, \ldots, 2$ compute:

$$\delta_i^l = o_i^l(1 - o_i^l) \sum_{j=1}^{s_{l+1}} w_{ij}^l \delta_j^{l+1} \tag{2.11}$$

4. Compute the partial derivative for each weight and bias:

$$\frac{\partial}{\partial w_{ij}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = o_j^{(l)} \delta_i^{(l+1)} \tag{2.12}$$

$$\frac{\partial}{\partial b^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \delta_i^{(l+1)} \tag{2.13}$$

Now, combining *gradient descent* and the *backpropagation algorithm* we can describe an algorithm to train our network:

1. Initialize the weights $\mathbf{w}^{(l)}$ and b^l to random values for every layer l.
2. Do steps 3 to 5 until the $Cost(\mathbf{W}, \mathbf{b})$ function is low enough or converges. This is referred to as an *epoch*.
3. Set $\Delta \mathbf{w}^{(l)} := 0$ and $\Delta b^{(l)} := 0$ for all l.
4. For i = 1 to m,
   (a) Use the backpropagation algorithm to compute $\nabla_{\mathbf{w}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$ and $\nabla_b^{(l)} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$ for every layer l.
   (b) Set $\Delta \mathbf{w}^{(l)} := \Delta \mathbf{w}^{(l)} + \nabla_{\mathbf{w}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$.
   (c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$.
5. Update the parameters:

$$\mathbf{w}^{(l)} = \mathbf{w}^{(l)} - \alpha[(\frac{1}{m} \Delta \mathbf{w}^{(l)}) + \lambda \mathbf{w}^{(l)}]$$

$$b^{(l)} = b^{(l)} - \alpha[\frac{1}{m} \Delta b^{(l)}]$$

2.1.3 Issues with object recognition

While the ANN model have proven useful in several applications, it falls short when it comes to object recognition in images. According to [8] there are three major reasons for this: .

1. **Topology**. A fully connected ANN does not take into consideration the topology of the input. An image has a strong 2D spatial locality correlation, which makes it possible to combine low-order features (edges, end-points etc.) in the same area into higher-order features (noses, ears etc.).

2. **Scalability**. Even small images contains a large amount of pixels/inputs, e.g. a 32×32 image contains 1024 pixels/inputs. A fully connected network with 100 hidden units would then end up with 1024×100 weights that needs to be calculated in the first layer. Thus making it harder to scale for larger images and rather inefficient .

3. **Object variance.**While objects are similar enough, on a higher level, to be grouped together into a class, they can still be very different on a lower level. E.g. a human face have several features that are needed for it to be defined as a face, e.g. eyes, mouth, nose etc. But the size and shape of these features tend to be very different from person to person. While it is possible for a standard ANN to compensate for these internal differences within a class, it would have to make three costly compensations. 1) The network would have to be very large, 2) it would probably contain several neurons with similar weight vectors positioned at different places in the network, and 3) it would require a massive amount of training samples.

2.2 Convolutional Neural Network

A *Convolutional Neural Network* [8] (CNN) is an extension of the *Artificial Neural Network* model, which is made specifically for object recognition in images or speech recognition. It was made in order to solve the issues that the classic ANN model faced.

2.2.1 Definition

The CNN model adds two additional types of layers, in addition to the standard ANN layers: a *convolution layer* and a *subsampling/pooling layer*. The idea behind the two new layers is to exploit the local 2D structure of images, i.e. pixels close to each other are highly correlated. By using local correlation one can extract and combine small local features (e.g. edges, corners, points) into

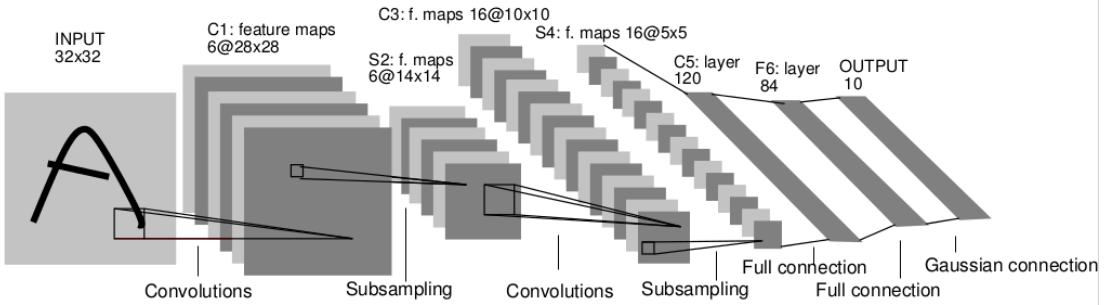

Figure 2.3: An example CNN, the LeNet-5 [8]. It consists of two convolution and subsampling/pooling layer pairs, which are connected to a fully connected ANN with 10 output classes.

higher-order features (e.g. a nose, a mouth, a forehead), which can in the end be recognized as an object (e.g. a face). A full network is illustrated in Figure 2.3.

Convolution layer

The convolution layer extracts a set of features from a set of input images. For each feature, the respective feature is extracted from all the input images and put in a feature map. E.g. if the filter extracts vertical edges, only the vertical edges from all the input images would remain in the resulting feature map. Thus different features can be extracted by having several feature maps that extracts different features.

The extraction is done by performing a *convolution operation* on the image, using a *kernel* that acts like a filter. The kernel is a 2D matrix that contains a set of weights. Depending on values of the weights, convoluting the image with the kernel will have wide range of effects, e.g. sharpening, bluring, edge detection and feature extraction. By training our network we can configure the weights to extract the features we need in order to recognize our desired classes.

After the convolution operation has been performed, a bias is added to every element in the feature map and the result is sent through a non-linear function, e.g. the hyperbolic tangent.

Formally we can define the convolution layer as follows. The layer accepts n images $X_1, X_2, \ldots, X_n$ as inputs, and produces m feature maps, $F_1, F_2, \ldots, F_m$. These feature maps are produced using a set of m learned kernels $W_1, W_2, \ldots, W_m$. Each feature map F_t is then produced by computing:

$$\mathbf{F}_t = Tanh(b_t + \sum_{i=1}^{n} \mathbf{X}_i * \mathbf{W_t}) \tag{2.14}$$

Where **F** is the resulting feature map, X is the input image, **W** is the kernel matrix, and b is the bias. $X * W$ is the convolution operation, which is defined as:

$$y_{ij} = \sum_{q=1}^{k} \sum_{p=1}^{k} x_{i+q,j+p} w_{qp} \tag{2.15}$$

Where x_{ij} is a value of the input matrix, w_{mn} is a value in the $k \times k$ kernel matrix, and y_{ij} is a value of the output matrix.

E.g. consider the LeNet-5 in Figure 2.3, in the first layer C1 the input is a single 32×32 image which is convoluted with 6 kernels, producing 6 feature maps. Thus $n = 1$ and $m = 6$. The resulting feature maps are then further processed by a subsampling/pooling layer S2 (see next section), which are used as input to the next convolutional layer C3. The six processed feature maps are then convoluted with 16 kernels, producing 16 new feature maps. Thus in this layer $n = 6$ and $m = 16$.

This helps solve the first two issues from Section 2.1.3. The neurons in a feature map share the same kernel, thus the same weights, which greatly reduces the size of the network. The convolution operation applies a 2D filter on the image, which makes the network able to exploit the spatial correlation in the image.

Subsampling/pooling layer

Once a feature has been detected, the exact position become less important. For example, the distance between the mouth and the eyes tend to vary between persons. So in order to make the CNN not too sensitive to the relative placement of features, the accuracy of the all feature maps needs to be reduced. This can be done by subsampling (i.e. partitioning) the feature map into $s \times s$ non-overlapping submatrices, and then perform a pooling operation on each respective matrix. There are two types of pooling operations which are used for CNNs:

- *Max-pooling* extracts the maximum value of the submatrix.
- *Average-pooling* extracts the average value of all the elements in the submatrix.

Given an output of m feature map inputs, each output matrix can be defined as:

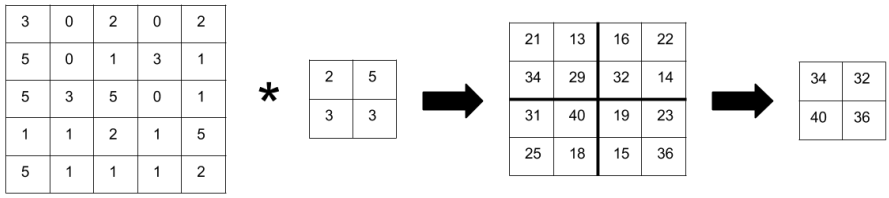

Figure 2.4: Illustration of the convolution and subsampling/max-pooling operations. The leftmost matrix is convulted with a 2×2 kernel, and the resulting matrix is subsampled into four non-overlapping areas where the max value is extracted.

$$\mathbf{O_t} = Tanh(b_t + subsample\_pool(\mathbf{F_t})) \tag{2.16}$$

Where O_t is the t'th output matrix, b_t is t'th bias, and F_t is the t'th input feature map, and the *subsample_pool()* function's operation is defined as either:

$$o_{ij} = max(x_{i\times s+p, j\times s+q}) \qquad q, p \in 1, 2, \ldots, s \tag{2.17}$$

or

$$o_{ij} = \frac{1}{c} \sum_{p=1}^{s} \sum_{q=1}^{s} x_{i+p, j+q} \tag{2.18}$$

Where o_{ij} is a value in the output matrix and f_{ij} is a value in the feature map, c is a trained constant, and s is the dimension of the subsampling size. A max-pooling operation is illustrated in Figure 2.4.

Thus, the subsample/pooling layer helps solve the two last issues from Section 2.1.3. By reducing the accuracy, the network is less sensitive to the difference between instances of a class. This also causes the network size to be smaller, since it does not require neurons to recognize the differences.

Figure 2.4 illustrates the convolution operation and the subsample/max-pooling operation, while Figure 2.5 illustrates the full operation of the convolution and subsampling/pooling layers.

2.2.2 Training

As mentioned, a CNN consists of three types of layers: a convolution layer, a subsampling/pooling layer and fully connected layer. The latter is trained as described in Section 2.1.2, using backpropagation and gradient descent. The two

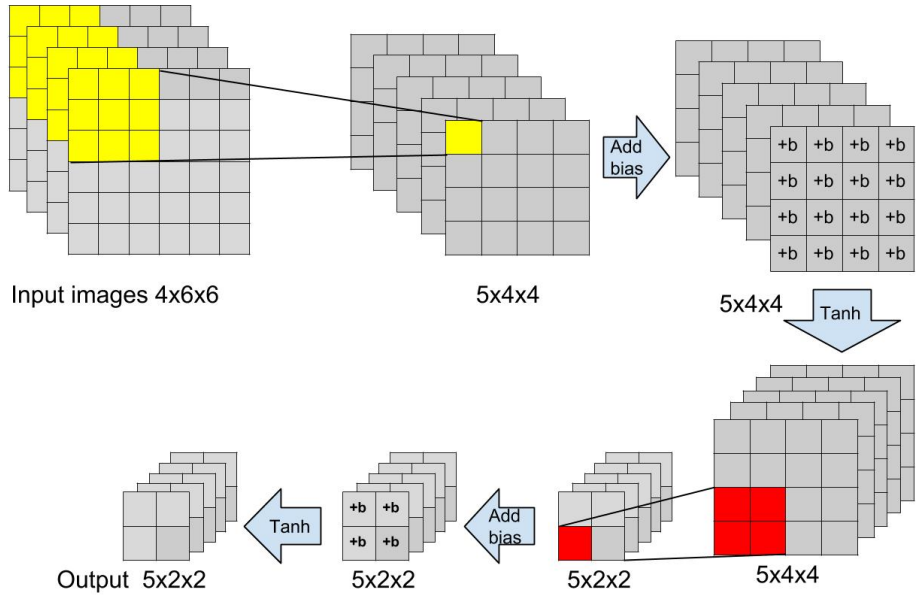

Figure 2.5: A visual overview of the operations performed by the convolution layer and subsampling/pooling layer with four input images. Yellow represents the convolution operation, and red represents the subsampling/max-pooling operation.

other layers use the same general algorithm, but the error δ^l and the gradient of $Cost(\mathbf{W}, \mathbf{b}; x, y)$ is calculated differently.

Since the backpropagation aglorithm starts at the last layer and work its way backwards, the error is first calculated for the fully connected layers. It is then provided to the subsampling/pooling layer, and finally to the convolution layer. Thus we first need to calculate the error for the subsampling/pooling layer, so we can propagate it to the convolution layer.

The subsampling/pooling layer does not contain any weights, and can therefore not be tuned. Thus it only needs to propagate the error it receives. Depending on which pooling operation is used, there are two respective methods for this. For max pooling, the error is simply propagated to the neuron that was chosen as the maximum value, while the rest are set to zero. For average-pooling we have to distribute the error evenly between all the responsible neurons. We therefore define the function *upsample(...)*, which performs the correct propagation operation depending on the type of pooler.

We can now formally define how to calculate the error and the gradient by

simply replacing the equations in step 3 and 4 in the backpropagation algorithm with the following equations. For simplicity we assume that convolution and subsampling/pooling is done in a single layer l.

$$\delta_k^l = upsample((\mathbf{W}_k^l)^T \delta_k^{l+1}) \bullet f'(\mathbf{Z}_k^l) \tag{2.19}$$

Where $(\mathbf{W}_k^l)^T$ is the weight matrix in layer l, δ_j^{l+1} is the error matrix for layer $l+1$, $\bullet$ is the element-wise product (i.e. Hadamard product), $f'(\mathbf{Z}_k^l)$ is the matrix containing the derivative of the activation function, and k indexes the filter number. I.e. it contains $o_{kij}^l(1-o_{kij}^l)$ for every neuron at index ij in feature map k in layer l.

Using this we can calculate the gradient:

$$\frac{\partial}{\partial \mathbf{w}_k^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \sum_{i=1}^{m} (\mathbf{o}_i^{(l)}) * \delta_k^{(l+1)} \tag{2.20}$$

$$\frac{\partial}{\partial b_k^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \sum \delta_k^{(l+1)} \tag{2.21}$$

2.3 Potential for parallelism

A vast amount of the computation required by a CNN can be parallelized. Thus, in order to achieve the processing of the network it is important that these potential parallelizations are identified and exploited. The most obvious being:

1. The convolution of a matrix $n \times n$ using a $k \times k$ kernel consists of $(n-k+1) \times (n-k+1)$ convolution operation, which each can be done in parallel. Thus convoluting the whole matrix could potentially take only the time it takes to perform one convolution operation.

2. The subsampling/pooling operation can also be parallelized by pooling all of the individual submatrices at the same time.

3. The computation of each of the individual feature maps and their corresponding subsampling/pooling. Which [13] referred to as *inter-parallelism*.

4. It is also possible to parallelize the computation of the feature maps that take more than one matrix as input. This is the case in the subsequent layers after the first. Which [13] referred to as *intra-parallelism*.

5. The activation of each neuron in the fully connected layer. One option is to parallelize them by creating a binary tree multiplier, where you have n units compute the product of the input and its respective weight, then you

use $\frac{n}{2}$ units to add two of the results each, and so on until you have a single value. This will reduce the time it takes from n time to $log_2 n$ time if they can all be done in parallel.

2.4 ZedBoard

We choose the ZedBoard as the development board for our implementation of our accelerator prototype. It contains a *Xilinx Zynq-7000 All Programmable System-on-Chip (SoC)* Z-7020, which consists of a dual-core *ARM Cortex-A9 MPCore based Processing System (PS)* and an Artix-7 XC7Z020 FPGA as *programmable logic* (PL) [14]. The PS includes on-chip memory, external memory interfaces, and a number of I/O peripherals. The system offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and *application specific standard product* (ASSP) [15].

The PL makes use of the second version of the *Advanced eXtensible Interface* (AXI4) bus protocol, which is part of the *ARM Advanced Microcontroller Bus Architecture* (AMBA) [16]. There are three types of AXI4 interfaces:

- AXI4 - for high-performance memory-mapped requirements.
- AXI4-Lite - for simple, low-throughput memory mapped communication.
- AXI4-Stream - for high-speed streaming data.

As long as a component in the PL implements any of these interfaces, it can be connected directly to the PS through a set of AXI4 bus ports. For the rest of this report we will refer to AXI4-Lite as AXI, and AXI4-Stream as AXIS. The prefixes S_ and M_ will refer to slave and master port, respectively. These acronyms will be use extensively in Section 4.4.

The feature that makes the ZedBoard especially well-fitted for hardware acceleration applications, is the tight coupling between the PS and the PL. The ARM processors can be connected directly to any component in the PL area through the *extended multiplexed I/O* (EMIO) port or a set of general purpose AXI ports (see Figure 2.6). In addition, there are four high performance (HP) AXI4 ports that PL components can use to access external memory directly. At max capacity the HP AXI4 ports have a bandwidth of 1200 MB/s. This integration of the PS with the PL allows levels of performance that the two-chip solutions (e.g. an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency and power budgets [15].

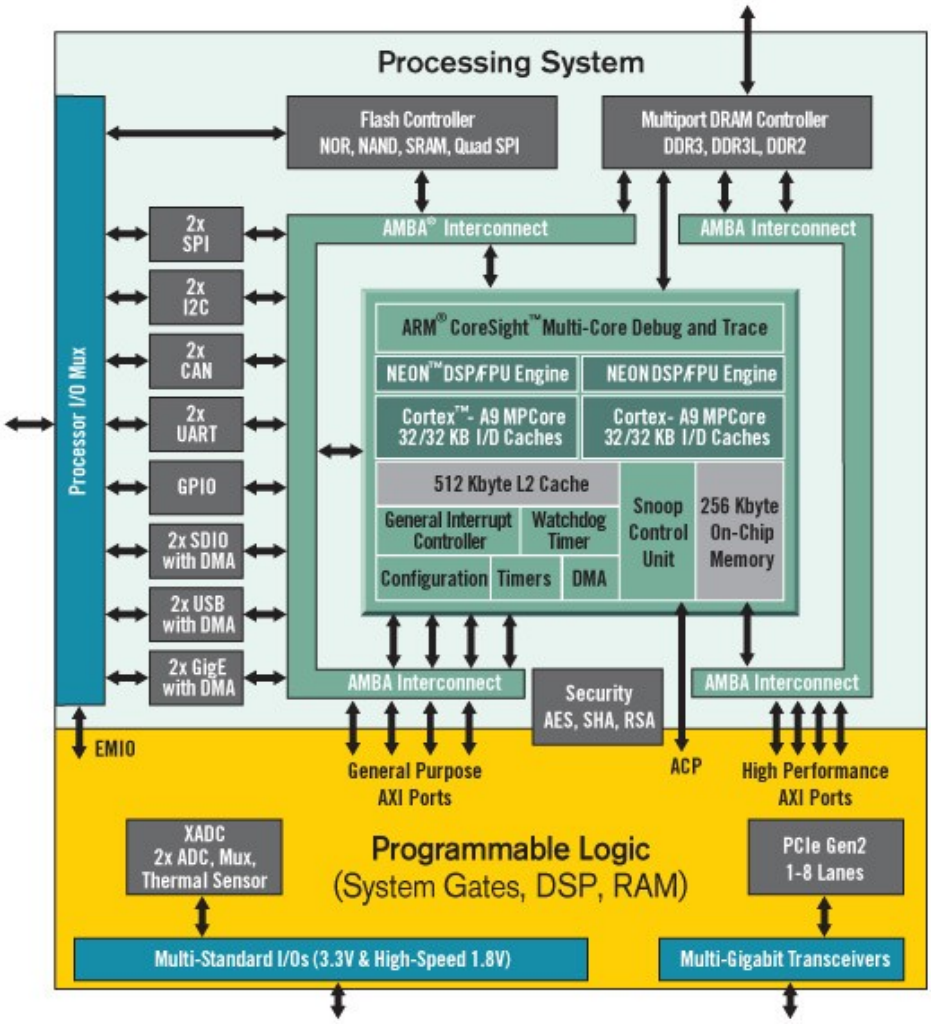

Figure 2.6: Zyng-7000 system architecture.

Chapter 3

Related Work

This section will give an overview of the current state of research on Convolutional Neural Networks.

3.1 Convolutional Neural Networks

The mathematical fundamentals for Convolutional Neural Networks was introduced as early as in the 1980s by Kunihiko Fukushima[17][18], in form of the *neocognitron* model. The model was later improved in 1998 by Yann LeCun, Lon Bottou, Yoshua Bengio, and Patrick Haffner - who introduced the *Convolutional Neural Network* model. In 2003 the model was simplified by Patrice Simard, David Steinkraus, and John C. Platt [19], in an attempt to make it easier to implement. The paper also mentions two of the main issues with CNNs: the size of the training set and the time spent training. In order to achieve high enough accuracy a CNN requires thousands of training samples, which needs to be labeled. Processing all of these samples and fine-tuning the networks takes a great amount of processing power, causing training to take days or weeks. These issues caused CNNs not to gain popularity before mid-2000.

The rise of the Internet, digital cameras, and Big Data have provided us with vast amounts of images which can be used for training. Improvements in the speed and sophistication of computer hardware have reduced the training time from days/weeks to hours. E.g. [20] purposes a GPU implementations which reduced the epoch (see Section 2.1.2) training time from 35 hours to 35 minutes. This demonstrates that highly parallel hardware vastly increases the efficiency of neural networks compared to CPUs.

These recent advancements have renewed the interest in neural networks and increased the research done on the field. As a result CNNs have become a leading

model within pattern recognition for computer vision. This can be illustrated by the fact that CNNs implementations have won several pattern recognition contests in the period 2009-2012, including IJCNN 2011 Traffic Sign Recognition Competition[21] and the ISBI 2012 Segmentation of Neuronal Structures in Electron Microscopy Stacks challenge[22].

3.2 Convolutional Neural Network in Hardware

There have been several proposed hardware architectures during the last decade, and below we will describe the more recent and relevant architectures. If the reader is interested in even older implementations, one can refer to [23], [24], [25], [26], and [27]. We have divided the architectures into two categories, mobile co-processors and server co-processors. The first is small architectures that are intended to fit into resource constrained environments, i.e. mobile applications, while the second is larger architectures that have virtually no resource constraint. But a common design goal for both categories are power efficiency.

Mobile co-processors
In [2] a CNN was implemented on a Virtex-4 SX35 FPGA from Xilinx. In this implementation all the fundamental operations were accelerated by a special-made ALU, and controlled by a 32 bit soft processor using macro instructions. That is, they created macro instructions for convolution, non-linear function, subsampling/pool and dot product between values at identical locations in multiple 2D planes and a vector. Training was done offline, and a representation of the network was provided to the soft processor. With this implementation they were able to process a 512×384 gray-scale image in $100ms$, i.e. 10 frames per second. The design was intended for use in low power embedded vision systems, e.g. robots, and the whole circuit board used less than 15 W.

Farabet and LeCun later improved the mentioned architecture in [28]. In this design they added multiple parallel vector processing units and allowed individual streams of data to operate within processing blocks. They were able to achieve 30 frames per second using 15 W. In addition they predicted a planned ASIC implementation of the system would increase the processing speed and reduce the power to 1 W.

In [13] they explore how they can exploit the parallel nature of CNNs. They introduce two types of parallelism found in CNNs, *inter-output* and *intra-output*. The first one comes from the observation that each feature map and the corresponding subsampling/pooling computation can be done in parallel. This is easily seen in the first layer. The second one refers to that the convolution of several inputs are combined to produce one feature map (see Figure 2.5), where the individual convolutions can be done in parallel. This one is present in all of

the convolution layers after the first layer. They exploit these observations by purposing a dynamically configurable co-processor on a FPGA, which can switch between computing several different feature maps in parallel and processing several inputs to compute one feature map. By doing this they are able to fully utilize the parallel nature of a CNN and reduce the intermediate storage on the FPGA. Using a Virtex 5 SX240T FPGA with 1024 multiply-accumulate units they were able to outperform a 2.3 GHz quad-core, dual socket Intel Xeon, and a 1.35 GHz C870 GPU by 4x to 8x.

[29] presents an architecture they named the *nn-X*. For the implementation they used a Xilinx ZC706 platform, containing a Kintex-7 FPGA and two ARM Cortex-A9. They made a set of *collections* that contained acceleration units for the convolution and subsampling/pooling operations. Each collection also contained a data router which could route data to the accelerator units, or to other collections in order to share data. The convolution and subsample/pooling layer was procssed on the FPGA using the accelerators, while the fully connected layer was processed by the arm processors. The authors claim that this architecture is the fastest and most power efficient of all the purposed architectures for mobile processors, to date. It is able to perform up to 227 G-ops/s, using 8W.

[30] focuses one the challenges of memory bandwidth related to deep convolutional neural networks. They argue that while accelerators are fast, slow memory makes it difficult to saturate the accelerators with enough data. To combat this they purpose a memory access optimized routing scheme, where they reduce the number of times a input map has to be transfered from memory to the accelerator. A crucial point here is that in general the output map is the sum of several convoluted input maps. Thus if an accelerator is only able to compute one output map at a time, the input maps have to be transfered to the accelerator several times. This architecture reduces the amount of such transfers by having a DMA for every two accelerator, and making the DMA transfer the same data to both of its accelerators. The accelerators will either produce an intermediate results or a complete output map, depending on how many iterations it has run. There is a total of eight accelerators, four which are used to combine intermediate results into complete output maps, and four to compute intermediate results. Using this memory scheme they were able to decrease the memory access by 2x and increase the hardware utilization by 2x.

Server co-processors

Hardware acceleration of CNNs have also gained popularity within the data center field. The Internet and Big Data have made it viable to have servers that perform image classification, image recognition and natural language processing, using CNNs. Since the main expense of data centers are power usage, using specialized hardware accelerators that provides good performance at low power have gained

increased popularity. Thus recently there have appeared architecture suggestions for much bigger FPGAs than the previously mentioned, since size is mostly a problem for mobile applications.

One of the most prominent architectures is the one suggested in [1]. In this paper they present a detailed analysis of computing throughput and memory bandwidth utilization. Using the roofline model [31] they explored the design space in order to detect possible optimizations, including loop tiling, unrolling and pipelining. Based on these analysis they purpose a architecture for a hardware accelerator. What separates this architecture from the previous ones is mainly that the accelerator computes the whole layer in one go, instead of parts of the layer and combining them later. That is, all the input data of a layer is inputted to the accelerator, it computes, and outputs all of the output feature maps of that layer. Previous implementations have primarily accelerated parts of the layer, or one feature map at a time. This greatly decrease the off-chip traffic, which is said to be the main performance sink for CNN accelerators. Such an architecture requires an extensive amount of hardware resources, which is why they implemented it on a Virtex 7. The reward is a throughput of 61.62 GFLOPS, using 18.6 W.

Microsoft, who have experimented on using accelerators for CNNs in their data centers, recently purposed an architecture which exceeds the previous one. In [4] they present an hardware accelerator that fit into a Stratix V D5 FPGA, and that can be integrated in their severs. Again, the main optimization is preventing off-chip memory, which they achieve by using an on-chip data re-distributor, making them able to compute several layers in a row. While the previous mentioned architecture computed one layer at a time, this architecture computes several, boosting the performance by 3x compared to the previous. Thus the system is still slower than a GPU implementation on a Tesla K40, which is 6x faster, but the accelerator is at least 2x as energy efficient.

Chapter 4

Architecture

In this chapter we will present our suggested architecture for a hardware accelerated forward propagation of a Convolutional Neural Network used for recognizing handwritten digits. The architecture will be presented in a top-down approach, starting with the topology and dataset of the network, followed by an overview of the software used, and finally, the hardware architecture of the accelerator. The source files for the VHDL and software code used in this project can for the time being be found at [32]

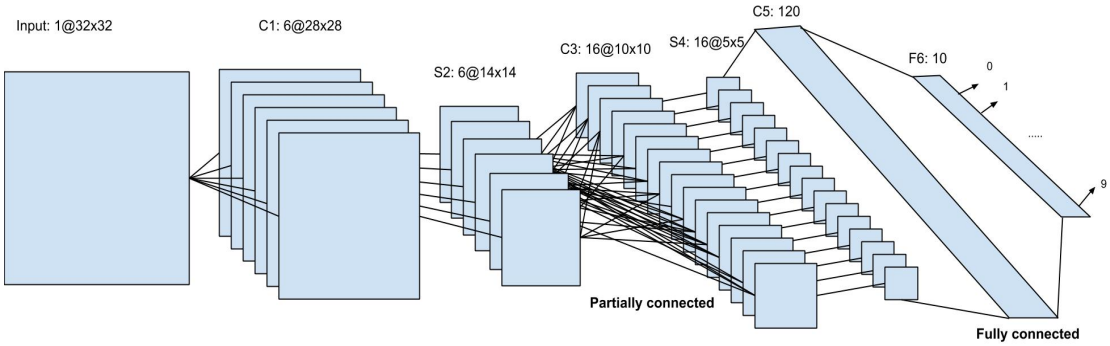

Figure 4.1: The topology of the implemented network.

4.1 Network Topology and Dataset

We chose to implement a network with a smiliar topology as the LetNet-5 for digit recognition, as seen in Figure 4.1. It consist of six layers:

- **C1, convolution layer**. Takes in a single 32×32 image of an digit. The image is convoluted using six different trained kernels, and outputs six respective 28×28 feature maps.
- **S2, subsampling/average-pooling layer.** Performs the subsample/average-pooling operation on each of the six 28×28 feature maps from the previous layer, using a respective trained value for each map. The resulting output is six 14×14 subsampled feature maps.
- **C3, partially-connected convolution layer.** Takes in six 14×14 feature maps which are partially connected to the sixteen 10×10 output feature maps. These connections are shown in Table 4.2. The connections specifices which inputs are needed to compute a given output. E.g. in order to compute feature map 13, input 2, 4 and 5 are to be convoluted with the 13's kernel. The respective convoluted inputs are then combined into a single matrice, where a bias and activation function is applied to every element - which give the resulting output feature map.
- **S4, subsampling/average-pooling layer.** Performs the subsample/average-pooling operation on each of the sixteen 10×10 feature maps from the previous layer, using a respective trained value for each map. The resulting output is sixteen 5×5 subsampled feature maps.
- **C5, fully connected convolution layer.** Takes in a sixteen 5×5 feature maps which are fully connected to the 120 1×1 output feature maps. Since the size of the output feature maps are a single value, the feature maps are basically standard neurons.
- **F6, output layer**. Takes in 120 neurons which are fully connected to the 10 output neurons. The output neuron with the highest value is the predicted value of the network.

There are three primary reasons for choosing this network. First, it is a relatively small network, which simplifies the implementation by reducing the chances of bugs and memory problems. Secondly, the kernel size of all the convolution layers are the same. This allowed for a less complex implementation, since we did not have to design our accelerator to support different kernel sizes, making it easier for the accelerator to support all the convolution layers. Thirdly, this network have been shown to work very well with the MNIST dataset, i.e. our own

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	o				o	o	o			o	o	o			o	o
2	o	o				o	o	o			o	o	o	o		o
3	o	o	o				o	o	o			o		o	o	o
4		o	o	o			o	o	o	o			o		o	o
5			o	o	o			o	o	o	o		o	o		o
6				o	o	o			o	o	o	o		o	o	o

Figure 4.2: Table showing which of the six feature maps from S2 that are needed in order compute the feature maps of C3.

experiments gave an accuracy of 99.1%. Since the aim of this project is exploring hardware acceleration, we did not wish to spend time finding a working topology for a given dataset. Using a topology that has been shown to give high accuracy allowed us to focus more on acceleration rather than topology theory.

As mentioned, we used the *the MNIST dataset*, available at [33]. It consists of 50 000 samples of handwritten digits ranging from 0-9, where 40 000 of the samples are used for training and 10 000 samples are used to determine the accuracy of the network.

4.2 What to accelerate

In order to decide which part of the network that should be accelerated one has to determine the most computational expensive part of the network. In the literature (see Chapter 3) there is a common consensus that the convolution layer is the most demanding layer, and [28] and [1] states explicitly that it amounts to about 90% of the total processing. We have confirmed this number in our own experiments, through a simple mathematical analysis of the network and by profiling a software implementation of the network.

Table 4.1 shows the number of connections for each layer in the network. Each connection corresponds to a multiply-and-accumulate (MAC) operation, e.g. 122304 MAC operations are required to compute C1. Since the number of activation functions to be computed is strongly correlated to the number of connections, we refrained from including them in the analysis. We see that 97% of the computations in our network is performed in the convolution layers, giving a clear indication of what layers should be accelerated.

Layer	Connections	Percentage
$C1$	122304	0.37
$S2$	5880	0.02
$C3$	151600	0.46
$S4$	2000	0.006
$C5$	48120	0.14
$F6$	120	0.004
$Total$	331104	1.0

Table 4.1: An overview of the number of connections in the network layers.

We also decided to accelerate the subsampling/pooling layers, even though only 0.8% of computations are done there. The reason for this that we were able to make a design where the subsampling/pooling could be done virtually in parallel with the convolution, at a minimal cost to hardware resources (see Section 4.4.6. We deemed the small cost worth the 0.8% potential performance boost. But more importantly, it makes our architecture eaiser to extend to compute several layers in a row, without going back to software, which would greatly reduce off-chip traffic and performance (see Chapter 6).

For future conveniences, we hereby define layer 1 as C1/S2, layer 2 as C3/S4, layer 3 as C5 and layer 4 as F6.

4.3 Software Architecture

This section gives an overview of software architecture used to compute the network and to control the accelerator.

4.3.1 Network Software

We have made extensive use of Taiga Nomi's C++ framework for neural networks, available at [34], in our project. The framework was used in order to train the parameters of our network, for measuring the efficiency of a pure software implementation, and as a basis for the implementation that uses the hardware accelerator. The framework treats each layer in the network as a separate software module, which makes it easy to swap different implementations of a layer. This simplified the process of integrating the hardware accelerator into the network, since we could simply exchange the original modules with our own.

Figure 4.3 A shows a simplified version of the architecture of the pure software implementation of our network. Each layer contains a set of pre-trained weights which are loaded before the network starts processing the images. When an image

is inputted to the first layer, it performs the calculations described in Chapter 2 in software, and propagates the result to the next layer.

Figure 4.3 B shows how the original software was changed in order to make use of the accelerator. As mentioned, we decided to accelerate the convolutional layer and the subsample/average-pooling layer, thus we wrote a new software module that would handle both operations. But instead of computing the operations in software, the new module transfers the input data and the weight to the hardware accelerator and extracts the result from the computations.

While our architecture supports accelerations of C5, we refrained from using it in its current form. The reason being that currently the accelerator is only able to compute one feature map at a time. Each computation comes with a certain amount of overhead, i.e. transferring data to/from the accelerator and configuring it. Thus for C5, which takes in 120 5×5 matrices, we figured that input was so numerous and so small that it would cause too much overhead in order to be efficient. In Chapter 5 we show data that confirms this claim.

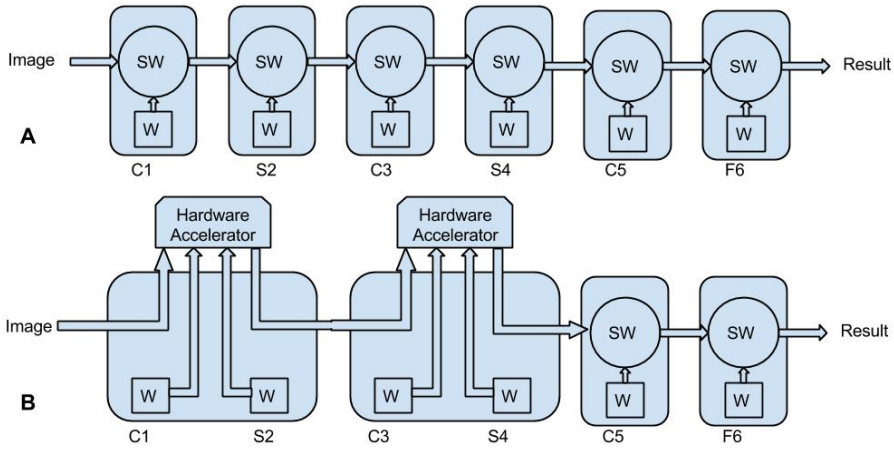

Figure 4.3: A simplified overview of the software architecture with and without hardware acceleration.

4.3.2 Hardware driver

A driver for the accelerator was written in order to create a simple and easy to use interface to the hardware. As mentioned, in its current form the accelerator is able to compute on feature map at a time, thus the input to the driver is all the data required to compute said feature map. That is, a set of images, their

respective kernels and bias, the average pooling constant and its respective bias. The driver then feeds this data to the accelerator, and returns the computed feature map. Due to the architecture of the accelerator the input has to be transferred in a certain order. The biases and average pooling constant first, the weights second and the image last. Figure 4.4 shows in what order the data should be in the accelerator FIFO buffer.

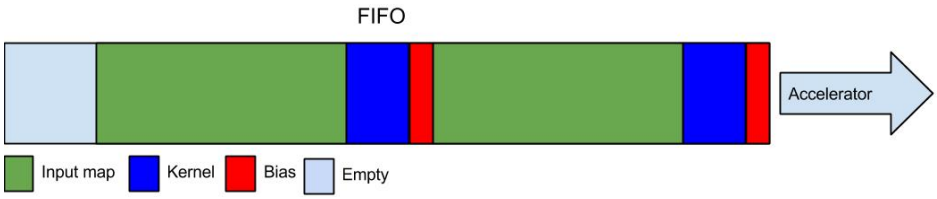

Figure 4.4: Shows how the data must be sequentially ordered in the input FIFO. In this example the output feature map is computed using two input maps.

For data transfer the driver uses a *direct memory access controller* (DMA) IP from Xilinx. This is where most of work on the driver had to be done, since the DMA interface is much more complex compared to the accelerator interface. The DMA is configured to transfer the weights and image(s) to the accelerator's input buffer, and extract the data from the output buffer. Since the output buffer is a FIFO, the DMA is able to extract each output value as they are produced, instead of waiting for the accelerator to finish and then transfer all the output data. Figure 4.4 shows how the data has to be structured when transferred to the accelerator's input buffer.

The processor can make use of the DMA by providing it with a set of *buffer descriptors* (BD), also called a BD ring. Each descriptor provides the DMA with the necessary information to perform a memory transfer: a source address and/or a destination address (depends on type of channel), and the length of the packet to transfer. One also need to set a *start of file*(SOF) and *end of file* (EOF) control bit for the first and last BD in the ring, respectively.

Our DMA has two channels, a transmit (Tx) and receive (Rx) channel. For the Tx channel the destination of the data is set to the address of the component connected to the DMA's Tx bus. Which in our case is the buffer which feeds the accelerator. Similarly, the Rx channel's source address is set to the component connected to the Rx bus, i.e. the FIFO buffer containing the output of the accelerator.

E.g. if a feature map requires two sets of input maps, weight and bias, then Figure 4.5 shows a simplified version of the Tx BD ring. Since the BDs are processed sequentially, the data will be transferred in the correct order, as in

Figure 4.4.

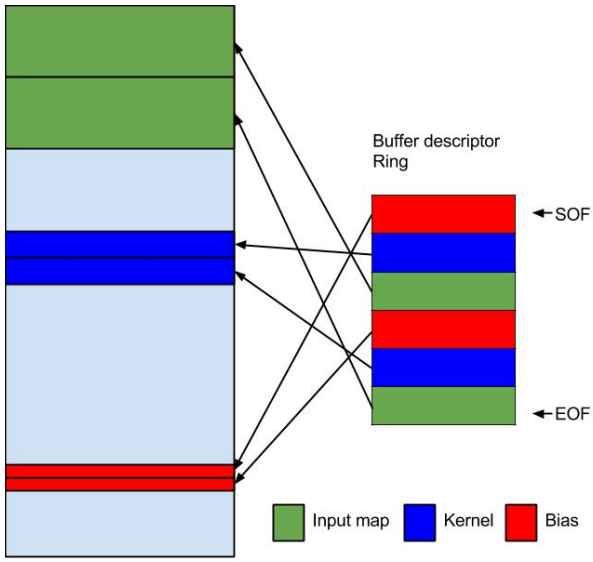

Figure 4.5: Simplified version of the Tx BD ring. First BD points to the location of the bias, second to the weights, and the third to the image, etc.

Xilinx provides a software driver for the DMA that manages the BD rings and initializes DMA transfers. A subset of the driver interface is listed below, showing the functions we used to control the DMA. For a more detailed explanation of the interface, please refer to the documentation in the driver source code, available at [32]

Listing 4.1: XAxiDma driver interface

```
/* Retrievies the config structure required to initialize the DMA
    . */
Config* XAxiDma_LookupConfig(u32 XAxiDma_id);

/* Initializes the DMA */
int XAxiDma_CfgInitialize(XAxiDma *dma, XAxiDma_Config *config);

/* Returns a pointer to the BD ring for the Rx channel */
XAxiDma_BdRing* XAxiDma_GetRxRing(XAxiDma *XAxiDmaInstPtr);

/* Returns a pointer to the BD ring for the Tx channel */
XAxiDma_BdRing* XAxiDma_GetTxRing(XAxiDma *XAxiDmaInstPtr);
```

```
/* Returns the maximum number of BDs given the available memory
    size */
int XAxiDma_BdRingCntCalc(u32 bd_size, u32
    size_of_available_memory);

/* Allocatetes memory for storing the BD ring */
int XAxiDma_BdRingCreate(XAxiDma_BdRing *RxRingPtr, u32 BaseAddr,
     u32 BdSize, u32 BdCount);

/* Allocated a number of BDs from the BD ring, which can be
    configured at submitted for hardware processing */
int XAxiDma_BdRingAllc(XAxiDma_BdRing *BdRingPtr, u32 BdCount,
    XAxiDma_Bd *FirstFreeBdInRing);

/* Sets of address the packet the BD should transfer. */
int AxiDma_BdSetBufAddr(XAxiDma_Bd *BdPtr, u32 addr);

/* Sets the length of the packet the BD should transfer */
int AxiDmaBdSetLength(XAxiDma_Bd *BdPtr, u32 length);

/* Set control signals for BD. E.g. SOF and EOF. */
XAxiDma_BdSetCtrl(XAxiDma_Bd *BdPtr, u32 CtrlMask);

/* Starts the channel. I.e. the DMA will start processing BDs as
    soon as they are submitted to hardware */
int XAxiDma_BdRingStart(XAxiDma_BdRing *BdRingPtr);

/* Submits the BD ring to hardware. Can not be accessed again
    before hardware is done processing. */
int XAxiDma_BdRingToHw(XAxiDma_Bdring *BdRingPtr, u32 BdCount,
    XAxiDma_Bd *FirstBd);

/* Extracts BDs from hardware after they are processed. Returns
    maximally BdCount BDs from hardware, but perhaps less */
int  XAxiDma_BdRingFromHw(XAxiDma_BdRing *BdRingPtr, u32 BdCount,
     XAxiDma *FirstBdRetrived);

/* Frees BdCount BDs after they have been processed by hardware.
    Must be done before the can resused for other transfers */
XAxiDma_BdRingFree(XAxiDma_BdRing *BdRingPtr, u32 BdCount,
    XAxiDma_Bd *FirstBdToFree);
```

Below we will show in simplified C code how we have used the above interface to configure and run, and transfer data to and from the accelerator.

The first thing that has to be done is to initialize the DMA driver, allocate memory for the Tx and Rx BD rings, and start the channels.

```
  void initializeDMA() {
    XAxiDma Dma;
    XAxiDma_Config *Config = XAxiDma_LookupConfig(DmaId);
```

```
  XAxiDma_CfgInitialize(&Dma, Config);

  SetupTx(&Dma);
  SetupRx(&Dma);
}

void SetupTx(XAxiDma *Dma) {
  XAxiDma_BdRingPtr *TxRing = XAxiDma_GetTxRing(Dma);
  int BdCount = XAxiDma_BdringCntCalc(MINIMUM_ALIGNMENT,
      TxBdMemorySpaceHigh-TxBdMemorySpaceBase);
  XAxiDma_BdRingCreate(TxRing, TxBdMemorySpaceBase,
      MINIMUM_ALIGNMENT, BdCount);
  XaxiDma_BdRingStart(TxRing);
}

void SetupRx(XAxiDma *Dma) {
  /* Virtually the same as SetupTx */
}
```

This should only be necessary to do once, as long as one does not need more BDs than what is currently allocated. Afterwards one can transfer data to the accelerator, run the accelerator and retrieve the output the following way:

```
void computeOutputMap(XAxiDma *Dma) {

  SetupTransferToAccelerator(Dma);
  SetupTransferFromAccelerator(Dma);
  WaitForTx(Dma);
  RunAccelerator();
  WaitForRx(Dma);

}

void SetupTransferToAccelerator(XAxiDma *Dma) {

  XAxiDma_Bd *Bdptr;
  int BdCount = NumberOfInputMaps*3; /* Multiply with three,
      because each input map needs to transfer three packets:
      bias, weights and image. */
  XaxiDma_BdRing *TxRing = XAxiDma_GetTxRing(Dma);
  XAxiDma_BdRingAlloc(TxRing, BdCount, &BdPtr);

  int index = 0;
  for each input map {
    for bias, weight and input map {
      XAxiDma_BdSetBufAddr(BdPtr[index], PacketAddr);
      XAxiDma_BdSetLength(BdPtr[index], LengthOfPacket);
      index++;
    }

  }
```

```
    XAxiDma_BdSetCtrl(BdPtr[0], SOF); /* First Bd in ring */
    XAxiDma_BdSetCtrl(BdPtr[BdCount-1], EOF); /* Last Bd in ring
        */

    XAxiDma_BringToHw(TxRing, BdCount, BdPtr[0]);

 }

void  SetupTransferFromAccelerator(XAxiDma *Dma) {
    int BdCount = 1; /* Only need to retrieve one output map */
    XAxiDma_bd *BdPtr;
    XAxiDma_BdRing *RxRing = XaxiDma_GetRxRing(Dma);
    XAxiDma_BdRingAlloc(RxRing, BdCount, &BdPtr);
    XAxiDma_BdSetBufAddr(BdPtr, PacketAddr); /* specifies where
        the output map should be stored */
    XaxiDma_BdSetLength(BdPtr, PacketLength);
    XaxiDma_BdRingToHw(RxRing, BdCount, BdPtr);
}

void WaitForTx(XAxiDma *Dma) {
  XAxiDma_Bd *BdPtr;
  XaxiDma_BdRing *TxPtr = XAxiDma_GetTxRing(Dma);

  /* Get processed Bds from hardware */
  int ProcessedBdCount = 0;
  while (ProcessedBdCount < TotalBdCount) {
    ProcessedBdCount += XAxiDma_BdRingFromHw(TxPtr, TotalBdCount
        , BdPtr);
  }

  /* Free all Bds when transfer is complete */
  XAxiDma_BdRingFree(TxPtr, ProcessedBdCount, BdPtr);
}

void WaitForRx(XaxiDma *Dma) {
  /* Virtually the same as WaitForTx */
}
```

The interface to the accelerator is designed to be as easy to use as possible, and requires little configurations. One simply needs to specify the layer that is going to be processed and the number of input maps that is needed to compute the output map. In addition, it is vitual that the input data is in the accelerators input buffer before it is started. With this is mind, the accelerator can be used in the following way:

```
 void RunAccelerator() {
       Xil_Out32(AcceleratorAddr+4, layer); //Set current layer
       Xil_Out32(AcceleratorAddr+8, nof_input_mapsx); //Set the
           number of input maps
       Xil_Out32(AcceleratorAddr, 0); //Start processing
 }
```

Where Xil_Out32(u32 addr, u32 value) writes *value* to the memory location of *addr* in the PL.

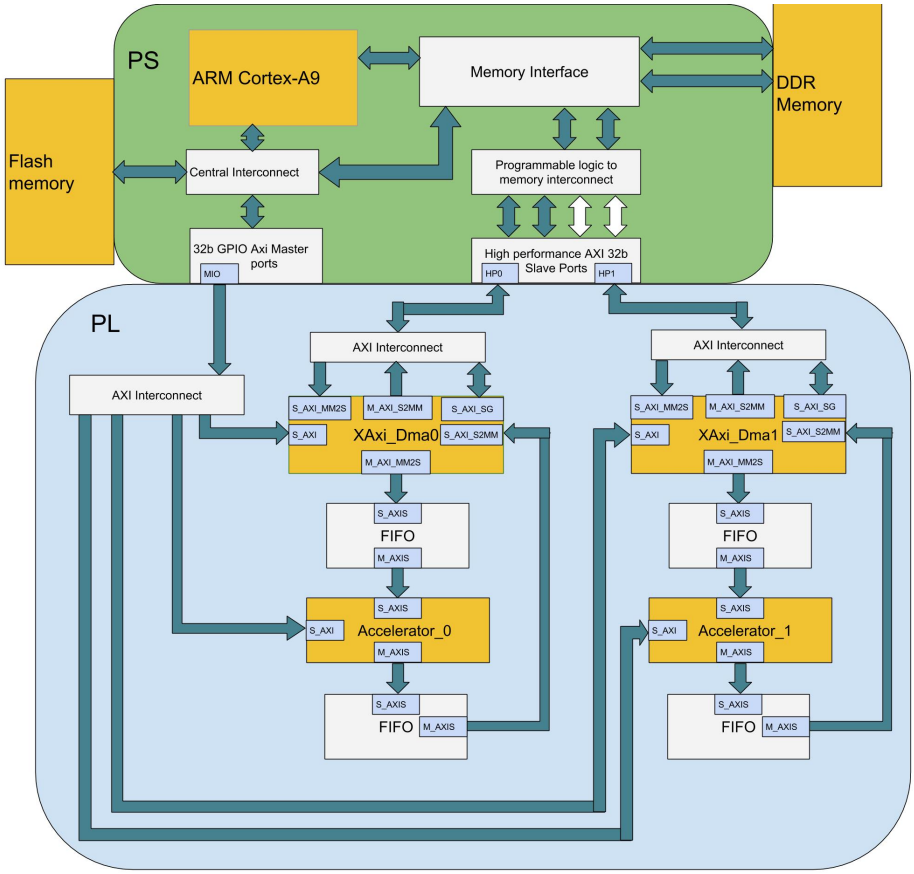

Figure 4.6: The system architecture.

4.4 Hardware Architecture

In this section we will describe the hardware architecture of the system as a whole, and more specifically, our accelerator. Do note that the descriptions and the figures are simplified to some extent. This simplification is done with the intent avoiding complex description and explanations. We will rather focus on

conveying the important ideas behind the design, instead of describing every tiny detail of the architecture.

4.4.1 Overview of Hardware Architecture

Figure 4.6 shows the block diagram of our system architecture. In this design we have to accelerators running in parallel, with their own respective DMA which is used to feed them data. The DMA's are connected to two of the four available *high performance AXI* ports, which are optimized for high bandwidth access from PL to external memory. The DMA has three bus interfaces that uses the HP ports:

- The AXI slave port *Memory-Mapped to Streaming* (S_AXI_MM2S) is used to transfer data from memory to the DMA. All the data transferred by the DMA's Tx channel is transferred to this port. The data can than be rerouted to any arbitrary hardware module, as long as they implement a S_AXIS bus port which is connected to the DMA's M_AXI_MM2S port.
- The AXI master port *Stream to Memory-Mapped* (M_AXI_S2MM) is used to transfer data from the DMA to external memory. The DMA receives data from the *S_AXI_S2MM* which it rerouted to the M_AXI_S2MM if the Rx channel is active.
- The *Scatter-Gather* (SG) is used to transfer BDs between software and hardware. I.e. when software is done configuring the BDs and submit them to DMA control, the DMA transfers the BDs from memory to itself, and starts processing them. When the DMA is done processing the BDs are returned to memory through the same port.

In order to configure and run the DMA, the processor is connected to it with a S_AXI bus. This is used to tell the DMA where in memory it can find the BD ring, when BDs are ready for processing etc.

The accelerators are connected to two FIFOs, one for input and the other for output. The DMAs fill the input FIFO with data which the accelerator consumes, and extracts the accelerator's output data from the output FIFO. The processor can configure the accelerator directly by writing to it via the S_AXI port.

The flash memory is used to store the trained parameters of the neural network, and the input images and their respective labels. When the application starts up, this data is parsed by the processor and then stored in the external memory, so it can be accessed by the DMA later on.

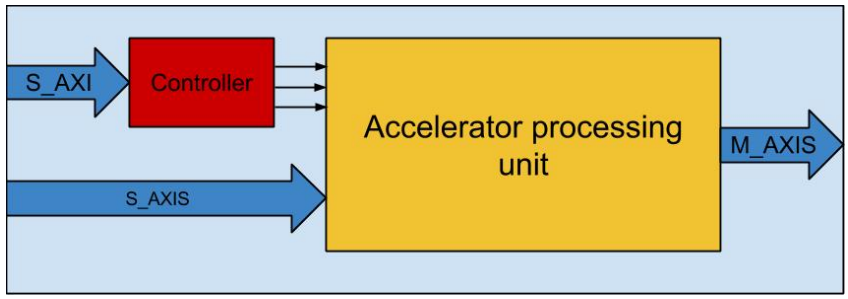

Figure 4.7: The interface of the accelerator.

4.4.2 Accelerator Interface

Figure 4.7 shows the block diagram for the interface of the accelerator. The S_AXI bus, which the processor uses to configure the accelerator, is connected to a controller. The input maps, weights and biases are streamed into the processing unit via the S_AXIS bus, and the resulting output map is streamed out via the M_AXIS bus.

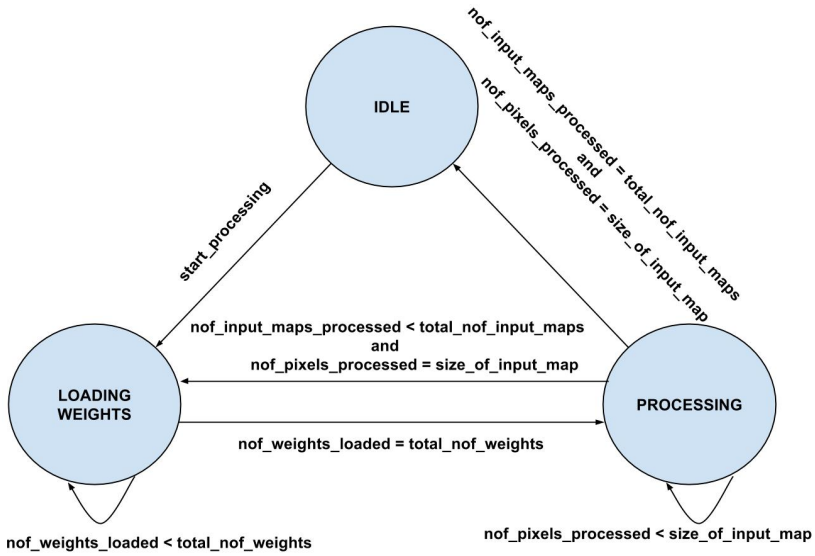

Figure 4.8: The three states of the accelerator's controller.

Layer	Input map size	Output map size	Kernel size
1	32×32	14×14	$5 \times 5 + 3$
2	14×14	5×5	$5 \times 5 + 3$
3	5×5	1×1	$5 \times 5 + 3$

Table 4.2: Hard-wired values stored in the controller that are specific to the network.

The controller is a state machine which controls the processing unit. It has three states: idle, loading weights and processing, as seen in Figure 4.8. The controller has two configurable registers, *layer_nr* and *nof_input_sets*. *Layer_nr* specifies which layer the output map corresponds to, and *nof_input_sets* specificities how many input maps are required to compute the output map. The controller have some hard-wired values which is related to the network it it supposed to processes. So based on the layer, it knows the size of the input map(s), output map and the kernel. E.g. for our specific network, the controller contains a table equivalent to Table 4.2. Layer 1 corresponds to C1/S2, layer 2 to C3/S4 and layer 3 to C5. The reason for adding three to the kernel size is to include the two biases and the average pooling coefficient.

The controller uses three control signals to manage the processing unit:

- *load_weights*. Tells the processing unit to extract data from the input buffer to fill its kernel buffers.
- *process_map*. Starts the processing of a single input map, extracted from the input buffer.
- *final_map*. Activated when the processing units starts processing its final input map, so it can output the result to the output buffer when done processing.

4.4.3 The Accelerator's Processing Unit

As previously stated, the accelerator takes n images as input, $I_1, I_2, \ldots, I_n$, n respective kernels $K_1, K_2, \ldots, K_n$, two biases and an average pooling coefficient, and outputs a single processed image O. Using the input images, the kernel and the bias, it performs the operations of the convolution and subsampling/average-pooling layer for a single feature map. Thus the output O is a subsampled/pooled feature map that has been produced by convoluting the images $I_1, I_2, \ldots, I_n$ with the kernels $K_1, \ldots, K_n$.

The accelerator can thus compute the whole convolution and subsample/pooling layer by doing the above computations for all the output feature maps in the

layer. One can exploit inter-parallelism by making several instances of the accelerator run in parallel. One can also exploit intra-parallelism, but then one need to connect the different accelerator instances so they can add up the results from the convolutions without using the intermediate convolution buffer, as described in [13]. Unfortunately, within the given time frame we were unable to get a system working that exploited inter- and intra parallelism. But the architecture is designed to be easily extendable to support this, given more development time.

The accelerators consists of five major components (Figure 4.9):

- **The convoluter**. Performs the convolution operation on the input.
- **The intermediate convolution buffer**. Since the resulting feature map is the sum of the convolutions of all the input images (with the exception of the first layer), this buffer is needed to store the intermediate results from the previous convolution, so that it can be accumulated with the current convolution. In the first layer of the network there is only one input image (i.e. $n = 1$), thus no summation is needed.
- **Tanh**. Performs the non-linear hyperbloc tangent function on the feature maps.
- **Subsample/average-pooler**. Performs the subsample/average-pool operation on the feature map.

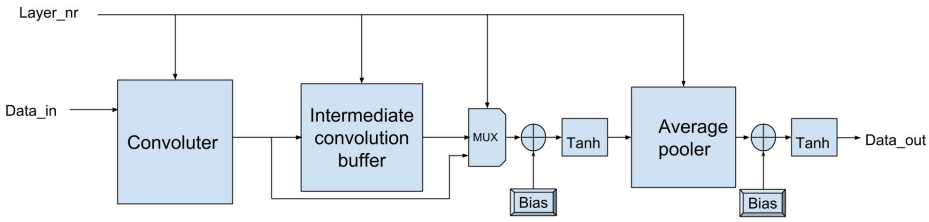

Figure 4.9: The architecture of the accelerator's processing unit.

The *layer_nr* signal is used to specify whether it is C1/S2 or C3/S4 that is being computed. The input image in the first layer is bigger than the images in the second (32×32 vs 14×14), which the convoluter and the average pooler need to (see Section 4.4.4 and 4.4.6). In addition in the second layer the intermediate convolution buffer needs to be activated so it accumulate and store all the convolutions needed to compute a single feature map. The mux is used to control which data to propagate to the average pooler, directly from the convoluter (C1/S2) or from the buffer (C3/S4).

In order to reduce resources spent and execution time, the accelerator uses Q16.16 fixed-point arithmetic, which is shown to give virtually the same network accuracy as floating-point arithmetic[35] [36] [37]. Something that has been confirmed by our own experiments. In order to implement fixed point arithmetic and fixed to float conversion we used the IEEE proposed libraries by David Bishop, available at [38].

In the following sections below we will provide a more detailed description of the convoluter, the hyperbolic tangent unit and the average pooler.

4.4.4 The Convoluter

This module is inspired by [2]. The input is a $n \times n$ image, and the output is a $(n-k+1) \times (n-k+1)$ feature map, using a $k \times k$ kernel. The kernel is stored in internal registers that must be rewritten for each different feature map that is to be computed. Every clock cycle, the module takes in a pixel as input, and after a certain delay it will output a processed pixel almost every cycle. Each pixel is inputted once, left to right, one row at a time.

It consists of 2D grid of multiply and accumulate (MAC) units which represents the convolution kernel. Thus the grid dimension is equal to the kernel dimension. In every MAC unit there is a register that contains the respective kernel weight. In every clock cycle the MAC units multiply the input pixel with its weight, and then accumulates the result from the previous cycle of the MAC unit to the left.

At the end of each row of MACs there is $n - k$ shift registers. The result of the last MAC in each row is stored in the first shift register, and the first MAC in each row takes the value of the last shift register of the previous row as accumulation input. The exception being the absolute first and last MAC unit. Every clock cycle, the values in the shift registers are shifted to the right.

By providing this delay you only have to input each pixel once during the convolution. Generally every pixel is needed for $k \times k$ convolution operations (the exception being the pixels close to the boarders of the image). Thus the shift registers are used to store the intermediate values of the convolutions until a pixel that is needed for the respective convolution operation is inputted.

The delay these shift registers cause are the reason for the delay before valid output pixels are produced. Thus from when the convolution starts, the output will not be valid before $k - 1$ rows of the image have been processed. And for every new image row, there will be a k cycle delay before the output is valid. The reason for this delay can be intuitively understood by remembering that the input image is a $n \times n$ matrix, while the output matrix is a $(n-k+1) \times (n-k+1)$ matrix.

Since the two layers in the network have different image sizes, but use the

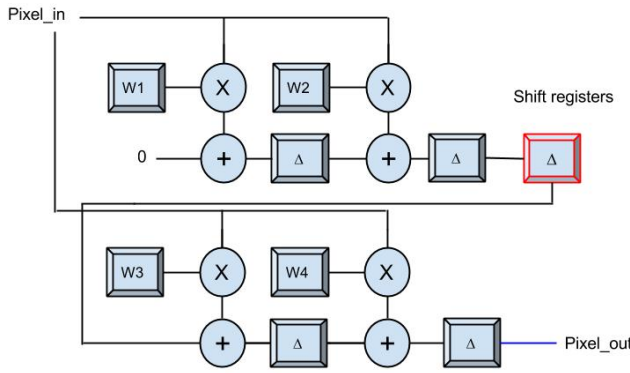

Figure 4.10: The Convoluter, when $n = 3$ and $k = 2$.

same kernel size, we can use the module for both layers. This is done by having the control signal *layer_nr* decide how many of the shift registers that are to be used during convolution. In the first layer all of the shift registers are used, but in the second only a subset is used. I.e. $n - k + 1$ of shift registers are used in each row, where n is either 32 or 14.

The loading of the weights takes $k \times k$ clock cycles, and the processing of the image takes $n \times n$ clock cycles. Thus the total number of cycles it takes to perform a full convolution of an image is $n \times n + k \times k$. But based upon the papers refered to in Section 3 it seems that n tends to be larger than k. E.g. for the first layer in the LeNet-5 [8], $n = 32$ and $k = 5$, the loading of the weights take 25 clock cycles and the image processing 1024 cycles. This means that the execution time of the convoluter is primairly bounded by the size of the image. But the size of the kernel decides the hardware resource cost of the module, since it requires $k \times k$ DSP slices on the FPGA.

4.4.5 The Hyperbolic Tangent

This module is based upon [39] using *piecewise linear approximation*. It takes as input a single value x and outputs a linear approximation of the hyperbolic function. Using a lookup table (Table 4.4) and the constants from Table 4.3 the module decides which linear approximation to use. In order to meet the timing constraints on the FPGA the module has a pipeline length of three.

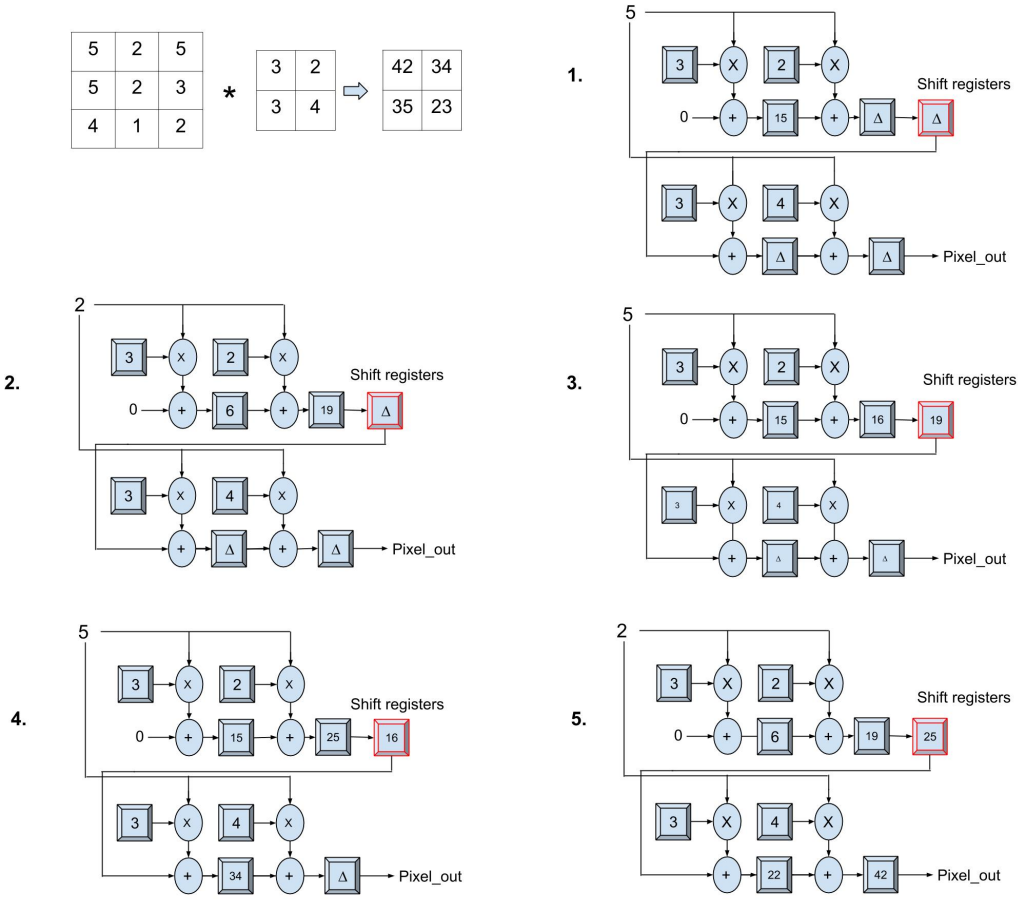

Figure 4.11: Example showing the five first clock cycle of an convolution. The weights of the kernel is already loaded into the MAC units, and every cycle a new pixel from the image in inputted. In the last example you can see that 42 is provided as the first output.

4.4.6 The Average Pooler

The average pooler performs the subsample/average-pooling operation described in Section 2.2.1. The input is a $(n-k+1) \times (n-k+1)$ feature map, and the output is a $(n-k+1)/p \times (n-k+1)/p$ subsampled/average-pooled feature map, where p is the dimension of subsample neighborhood.

Constants
$m_1 = -0.54324$
$m_2 = -0.16957$
$c_1 = 1$
$c_2 = 0.42654$
$d_1 = 0.016$
$d_2 = 0.4519$
$a = 1.52$
$b = 2.57$

Table 4.3: The constant used for the hyperbolic tangent approximation.

Conditions	Output
$0 \leq \|x\| \leq a$	$sign(x) \times [0.5 \times m_1 \times \|x\|^2 + c_1\ times \|x\| + d_1]$
$a \leq \|x\| \leq b$	$sign(x) \times [0.5 \times m_2 \times \|x\|^2 + c_2\ times \|x\| + d_2]$
$otherswise$	$signed(x)$

Table 4.4: The piecewise linear approximation of the hyperbolic tangent.

The average pooler performs basically two operations, pooling and averaging. The input is divided into $p \times p$ non-overlapping neighbourhoods, which are also refered to as pools. The pooling operation consist of simply summing the data within the respective neighbourhoods. The averaging operation is to multiply the summed pools with a trained average value, which produces a valid output. Figure 4.12 gives an overview of the average pooler architecture, where the SUM module and the shift registers are used for the pooling operation, while the trained C value is used to average the sums.

Since the input is a 2D matrice that is inputted one value at a time left to right, one row at a time, the average pooler will have to keep track of data from $(n-k+1)/p$ different neighbourhoods simultaniously. That is, after the average pooler has received p inputs from the first neighbourhood, it will next receive p inputs from the next neighbourhood, and so on until the end of the first row is reached. It will then receive data from the first neighbourhood again. This will continue until it has processed p rows, after which it will have processed the first $(n-k+1)/p$ neighbourhoods, i.e. a row of neighbourhoods. It can then continue with the next row of neighbourhoods.

In order to keep track of $(n-k+1)/p$ neighbourhoods at a time, the average pooler contains a set of $(n-k+1)/p-1$ shift registers which are used to store the intermediate sum of the neighbourhoods. The sum module keeps track of the current neighbourhood and accumulate the input data if it belongs to said neghbourhood. When a new neighbourhood is about to be inputted all the shift

registers are shifted one to the right, and the sum module extracts the value of the rightmost register.

The control unit keeps track of when to shift the registers and when a neighbourhood is fully summed. To do this the unit contains two counters, *row_num* and *column_num*. When a new pixel is inputted the *column_num* counter is incremented, and when it reaches the end of the row the *row_num* counter is incremented. Every time $column\_num \ mod \ p = 0$ another pool is encounted, and the shift registers are shifted one to the right. When $column\_num \ mod \ p = 0$ and $row\_num \ mod \ p = 0$ the final value in a pool has been reached, and the final sum is multiplied with the trained value C and outputted.

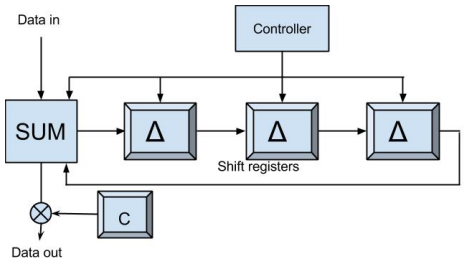

Figure 4.12: The average pooler. The summation module and the shift registers are used to sum up the respective pools. The trained value C is used to average the summed pools.

The execution speed of the average pooler module is bounded by the size of the feature map, $(n - k + 1) \times (n - k + 1)$ clock cycles, finishing one cycle after the last pixel has been inputted. Thus by streaming the output of the convoluter to the average pooler, both will finish only a few cycles apart, effectively running both jobs in parallel. The resource usage of the module is bounded by the size of the subsampling dimension, since it requires a number of shift registers equal to the size of the dimension. In addition is consumes one DSP, which is used for the averaging operation at the end. But essentially its resource usage is quite low.

Chapter 5

Results and Discussion

This chapter will present the results from performance testing our purposed architecture. Section 5.1 will give an analysis of the hardware resource consumption of our accelerator. In Section 5.2 we will describe the set up for our performance measurements, give an analysis of the performance of our system, and how the accelerator affect the different layer. We will also compare the performance of the system with a software implementation running on a laptop CPU.

Remember from Section 4.2 that we defined layer 1 as C1/S2, layer 2 as C3/S3, layer 3 as C5, and layer 4 as F6.

5.1 Hardware Resources

As mentioned in Section 2.4 we used a Zedboard, with an Artix-7 FPGa, for prototyping our suggested architecture. A short overview of the most vital resources available on the Artix-7 can be seen in Table 5.2. Table 5.1 shows the resource consumption of each accelerator as the number of accelerators increases. The "DSPs: Convoluter" column shows how many DSPs the covoluter module within the accelerator consumes.

As we can see from Table 5.1 the convoluter makes heavy usage of the DSPs. In its current state it contains 5×5 MAC units, which optimally consumes four DSPs each. When the number of available DSPs is reduced, the MAC units need to exchange the DSP with LUTs, which causes an massive spike in resource consumption.

Since the accelerators optimal number of DSPs is 116 out the of 220 available, increasing the number of accelerators from one to two does not cause any critical spike in resource consumption. But the converting of 12 DSPs to LUT logic is still significant, causing a total of 4000 more LUTs being used. It reaches critical

Nof. accelerators	Slice LUTs	Flip-flops	DSPs	DSPs: Convoluter
1	7986	3310	116	100
2	9941	3310	110	94
3	18699	3310	66	50
4	20971	3325	54	50

Table 5.1: Table showing how the resource usage varies for each accelerator as the number of accelerators increases.

Slice LUTs	53,200
Flip-flops	106,400
DSPs	220

Table 5.2: Available resources on the Artix-7 FPGA

levels when we increase the number of accelerators to three, leaving only 50 DSPs to be used by each convoluter. This causes each accelerator to consume 18699 LUTs, giving a total of 56097, which exceeds the maximum of available LUTs by 2897. Thereby limiting the number of accelerators to two, unless we change to a bigger FPGA. While a bigger FPGA, e.g. an Virtex 7, would provide us with more hardware resources, it would also increase the size of the chip and power consumption. Making it unsuitable for mobile applications.

This unexpected consumption of resources is unfortunate, since we would preferably wish to run at least four accelerators in parallel, in order to exploit the bandwidth from all four high-performance AXI4 ports. We have been unable to explore ways to reduce the convoluters consumption of DSPs due to time constraints, but it is definitely something that should be done in future work.

5.2 Performance

5.2.1 Setup

In order to determine the execution speed and power efficiency of our system we have compared it to the ARM Cortex-A9 CPU on the Zedboard and an ASUS X550JK laptop with a Intel Core i7 4710HQ CPU. Both CPUs ran the pure software implementation of the CNN, while our system used a combination of hardware and software, as described in 4. The clock speed of the accelerator was set to 100 MHz. We ran our own system with four different configurations:

- 0 layers. No acceleration. The ARM processor computes the whole network.

- 1 layer. The accelerator computes C1 and S2 (see Section 4.1, while the ARM processor computes the rest.
- 2 layers. Accelerating C1, S2, C3 and S4.
- 3 layers. Accelerating C1, S2, C3, S4 and C5.
- Two accelerators: 2 layers. Running two accelerators in parallel, both accelerating two layers.

In order to determine the energy efficiency of the different systems we used the metric *images/J*, i.e. number of images processed per Joule. We also included a metric for measuring execution speed, using images/second. Despite power efficiency being the main focus of this assignment, execution speed can be interesting for several applications and is closely related to power usage. Note that these images are 32×32, and thus processing one image corresponds to 331104 multiply-and-accumulate operations. Thus, if one wish to convert the metric images/second to operations/second, one simply need to multiply with that number.

The measurements were done by timing the processing of 10 000 images from the MNIST dataset, while measuring the power consumption.

Total board power was determined by measuring the voltage over pin 1 and 2 on J21 current sense resistor on the Zedboard during execution. We can then use the following equation to calculate the power consumption:

$$P = (V_{in} - V_{measured}) \times \frac{V_{measured}}{R} \tag{5.1}$$

Where V_{in} is the input voltage 12V, $V_{measured}$ is the measured voltage across the pins, which have a resistance of $R = 10m\Omega$.

With the FPGA programmed and the accelerator activated the board measured to 4.18 W, while the ARM processor alone measured to 3.82 W. The second core was turned off in both cases. We were unable to measure the power consumption of the laptop directly, and therefore used the power estimation provided by ASUS [40]. The laptop CPU uses 47 W, and including RAM, motherboard and various peripherals, we estimate a total consumption of 60W. Do note that this estimate is not completely accurate, and could potentially be lower.

5.2.2 Discussion

Analysis of the accelerator's performance

Figure 5.1 shows how accelerating the different layers in hardware affect the performance. We can see that accelerating one layer provides us with a almost

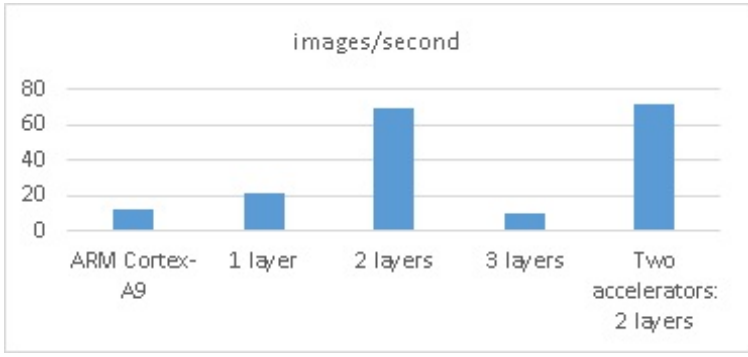

Figure 5.1: Overview of how the performance changes with different configurations of the system, when computing the whole network.

2x speed-up, while accelerating two layers give close to 6x speed-up. In stark contrast, accelerating three layers slows down the system and performs even worse than computing everything in software. We believe this is mainly due to how we move data to the accelerator. Each of the 120 outputs from C5 are computed using the same 16 input maps, but with a different set of 16 distinct kernels. This means that in the current state of our system, we transfer the same 16 input maps 120 times to the accelerator. We consider this a rather ineffective memory access scheme, and purpose a better one in Chapter 6.

Another interesting aspect of Figure 5.1 is that running two accelerators in parallel provides virtually no speed-up compared to only running one. There are two feasible explanations for this: 1) the accelerators get starved, i.e. they not fed data fast enough, and 2) the main bottleneck in now layer 3, which reduces the significance of improving performance for layer 1 and layer 2.

In order to determine which layer provided the biggest bottleneck, we decided to measure the performance when removing layer 3 and 4 from the network. I.e. only processing layer 1 and 2. The results can be seen in Figure 5.2.

Here we see that processing only the first two layers executes 4x faster than processing the whole network. This means that C5/F6 stands for 75% of the processing time of the whole network. Since C5 has 48120 connections and F6 has 120, it is clear that C5 is the main bottleneck.

This goes to show that accelerating layer 3 can provide a major boost to the performance of our computing. But, as mentioned, the current scheme we are using to accelerate layer 3 needs to be improved.

In addition, we can see that using the accelerator gives up to 51x speed-up when processing layer 1 and 2. Which demonstrates how much faster the accelerator is than the ARM processor.

Figure 5.2: Performance when removing layer 3 and 4 from the network. I.e. only processing layer 1 and 2.

But this does not explain why we get so little speed-up when running two accelerators in parallel. Ideally, we should get a 2x speed-up compared to running only one accelerator. In turns out that the DMA bug (see Section 4.3.2, which forces us to re-initialize the DMA after it is done processing a BD ring, is part of the explanation. This re-initialization causes the memory needed to store the BD rings for the DMA channels to be reallocated every time, and this takes time. In order to predict the performance if this bug was fixed, we measured the time it took re-initialize the DMA and subtracted it from the execution time. This produced the results seen in Figure 5.3.

Figure 5.3: Performance when processing only layer 1 and 2, and subtracting DMA initializations.

We can see that this increases the performance by more than 2x, and brings the configuration using two accelerators closer to being twice as fast as the one using one accelerator. But we can still see that there is something that is preventing the system to fully exploit having two accelerators running in parallel. We suggest two probable reasons for this:

1. Currently the processor has to wait for the data transfers to and from the accelerator to finish, without doing anything productively. This causes the accelerators to be starved, since the processors have to set up the transfers

while the accelerators do not have to data to consume. Preferably it could set up the other transfers while the accelerator is processing and/or the DMA performs transfers, but due to the initialization bug we have been unable to implement this.

2. Another fault of the design is that the input buffer to the accelerator has to be filled before it can start processing. A better solution would be to simply stream the data into the accelerator, and make stall when data is not available.

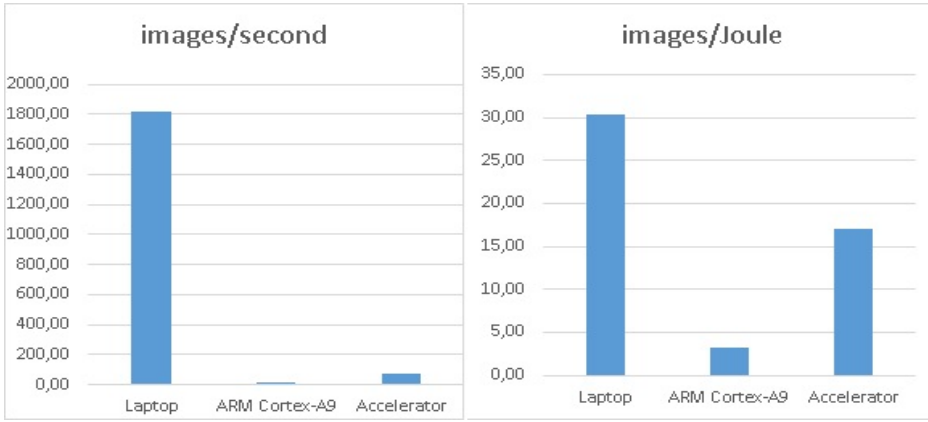

Figure 5.4: Performance when computing the whole network.

Comparison to a Intel Core i7 4710HQ CPU

In Figure 5.4 we can see how our system performs compared to the laptop CPU and the ARM Cortex-A9, when computing the whole network. In this comparison we used our best performing configuration of the system, two accelerators in a parallel that computed layer 1 and 2, while layer 3 and 4 was handled by the ARM processor.

In the current state of the system, it gets handily outperformed by the laptop. The laptop is 25x faster than our system, and almost 2x as power efficient. The reason for this is the bottleneck created by layer 3 and 4, which is computed by the ARM processor.

If we instead of comparing the performance of the whole network, compare the performance when computing layer 1 and 2, our accelerator do much better. As we can see from Figure 5.5, our accelerator is now 7.5x times slower than the

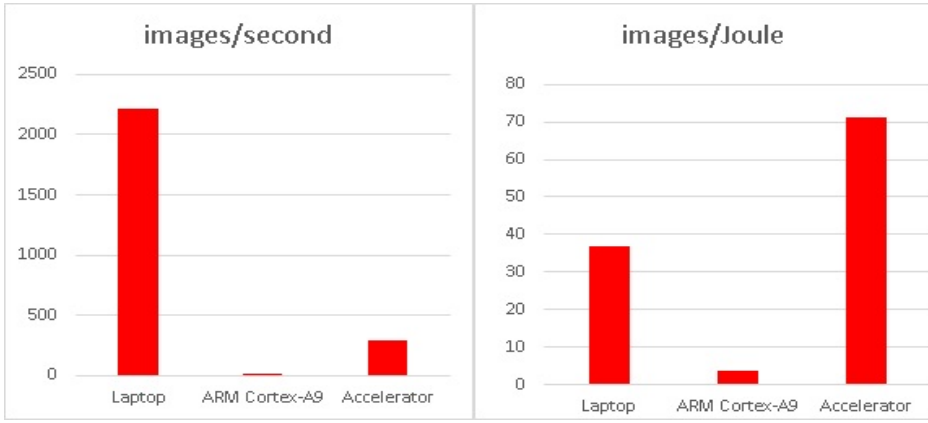

Figure 5.5: Performance when only processing layer 1 and 2.

laptop, but 2x as power efficient. If we also subtract the performance loss caused by the DMA initialization bug, our accelerator become 5x as power efficient as the laptop. This basically means if our accelerator is going to compete with a state of the art CPU, it has to be extended to it can effectively compute layer 4.

But despite it not being able to out-perform a laptop CPU, our accelerator would still do better on mobile platforms with power budgets. Our whole system uses 4.18 Watt, while the laptop CPU itself uses 47 Watt. It should also be noted that our system is only a prototype, and there are multiple optimizations that can be done in order to boost performance. Chapter 6 will discuss these optimizations.

Chapter 6

Future work

Developing a convolutional neural network hardware accelerator is a complex and time consuming task. There are thus several performance related improvements that we would wish to have implemented and tested, but sadly we ran out of time. In this chapter we will give an overview the planned, but unfinished, features we would wish to extend to our current architecture. The features are listed in a priority order, and we will give some indication of how much work is required to implement said features.

1. **Use both of the ARM processors, instead of just one.**

   Computing a CNN is an embarrassingly parallel task, and should thus be easily extendable to make use of both the ARM processors. A potential scheme is that each of the processors is designated an DMA and accelerator, which they can use to compute the network. Doing this can potentially provide a 2x speed-up.

2. **Improve the memory transfer and access scheme.**

   As we mentioned in Section 5.2.2, the current memory access scheme is inefficient. This is because we transfer the same input maps multiple times to the accelerator, since different output maps require the same input maps. A better scheme would be to have two input buffers to the accelerator, one for input maps and one for weights/bias. One could then transfer all the input maps to the map buffer once, and let them reside there through the computations of the whole layer. The accelerator could then access the required input maps for the respective output map directly from that buffer. Thus would reduce off-chip traffic greatly, since the input maps are only transferred ones. It would require some extra control logic, so the accelerator accesses the current input maps in the buffer.

3. **Stream data through the accelerator, instead of filling the buffer first.**

   Currently all the data required for computations are loaded into a buffer before it is processed by the accelerator. Changing it so that the data can be streamed directly into the accelerator would provide two potential benefits: 1) faster processing, since data can be processed while the DMA is transfering data to the accelerator, 2) reduced storage on chip, since we no longer need to store all the data in an input buffer. Should be fairly easy to extend the design for this, but due to time constraints, it was not implemented.

4. **Stay in hardware, instead of going back to software for next layer**

   The main reason [1] and [4] achieved high performance was by reducing off-chip traffic. In the current state of our system, software has to be involved for every feature map to be computed, and data is being transferred back and forth between software and hardware several times. This is inefficient. Thus extending the system with logic that can redistribute the output maps as new input maps without involving software could provide a substantial performance boost. But it will increase resource usage and development time, and will probably require a bigger board. Both the mentioned papers used FPGAs at the size of a Virtex 7. The mentioned improved memory scheme is a step in this directions.

5. **Explore ways to reduce the resource usage.**

   The primary focus of this project has been to get the prototype up and running, and little thought have gone to examine ways to minimize resource consumption. Currently we only have enough resources to fit two accelerators on the board. Thus if we wish to extend the design and/or run several accelerators in parallel, we either need to change to a bigger board or minimize the resource usage of our design. Any future work would do well to explore this area.

6. **More accelerators in parallel.**

   Currently we are only running two accelerators in parallel, which both have a respective DMA that moves their input and output data. The maximum DDR bandwidth is at about 3.2 GB/s, and each DMA has access to a *high-performance* port which can deliver up to 800 MB/s. This effectively means that we are exploiting half of the available DDR bandwidth. Given a big enough hardware platform or improvements to resource usage, as mentioned in the above point, this feature should be simple to implement. But before this can be exploited, the memory access scheme needs to be improved so the accelerators do not get starved.

7. **Hardware accelerate float to fixed.**

   Currently our system pre-processes the image and weights into fixed point before processing them. If this system were to be integrated into system using floating points it would be beneficial to do this transformation in hardware. Currently we are able to do fixed point to floating point in hardware using only one clock cycle, and thus we believe it should be possible to do the same for float to fixed.

8. **Test with bigger images.**

   In Chapter 5 we only compare the performance when computing a relatively small network, i.e. the input is a 32×32 image. In other CNN applications, images tend to be much bigger (see Chapter 3). This could impact the performance difference between our accelerator and the i7 CPU. I.e. with bigger images that do not fit in the i7 cache, our accelerator may be relatively faster than what is shown in Chapter 5. This should therefore be tested.

Chapter 7

Conclusion

In this report we have presented our hardware accelerator prototype for a Convolutional Neural Network. We have given a thorough description of our whole system, and how it was implemented on a ZedBoard development board. We have demonstrated the performance of our network by using it to compute a LeNet-5 inspired network, in order to classify handwritten digits. Compared to an ARM Cortex-A9 we achieved a significant performance boost, and outperformed an i7 CPU on the two first layers of the network in terms of energy efficiency.

We set out to complete two mandatory and two optional tasks. We will here give a quick summary of which of them were completed, and which ones remain incomplete:

Task 1 ***(mandatory)*** Implement a hardware accelerator for a Convolutional Neural Network, with the intention of improving energy efficiency. **Completed.**

We have in this project built a full system that is able to compute a CNN using a handcrafted hardware accelerator. Chapter 4 gives a complete description of its design and implementation, and Chapter 5 shows it improves energy-efficiency by 5.6x over an ARM Cortex-A9 processor.

Task 2 ***(mandatory)*** Compare our accelerator to an equivalent pure-software implementation on a general-purpose CPU, primarily in terms of power consumption. Chapter 6 gives further suggestions on how this design's performance can be further improved. **Completed.**

Chapter 5 gives an detailed analysis of how our system performs in terms of both execution speed and performance. While our accelerator is more power efficient

on the layers it was designed to enhance, it gets held back by the bottleneck created by the layers that are not accelerated. Because of this our system is notable to outperform a state of the art CPU. But the results provided gives a strong indication with further improvements, it has the potential to do so.

Task 3 ***(optional)*** Implement said system on a Zynq FPGA board, but weigh the advantages and disadvantages of other platforms, such as SHMAC or other FPGA platforms. **Partly completed.**

Our design was implemented on a ZedBoard, which contains a Zynq FPGA board. Section 5.1 provide some argumentation for using a bigger board, since it would allow us to run more accelerators in parallel. But apart from that we have not made any considerations of moving to another platform. This is mainly due to the amount of work that had to be done to get our current design working, and find ways to further optimize the design.

Task 4 ***(optional)*** Extend the system to be able to recognize objects from a web-cam stream. **Not completed.**

Due to time constraints we were unable to complete this task.

Bibliography

[1] Chen Zhang, Yijin Guan, Peng Li, Bingjun Xiao, Guangyu Sun, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170, 2015.

[2] Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. CNP: An FPGA-based processor for Convolutional Networks. *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, 1(1):32–37, 2009.

[3] Shuiwang Ji, Ming Yang, and Kai Yu. 3D convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–31, 2013.

[4] Kalin Ovtcharov, Olatunji Ruwase, Joo-young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. Technical report, Microsoft Research, 2015.

[5] Eric S. Chung, Peter a. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 225–236, 2010.

[6] M Duranton, K De Bosschere, and J Maebe. The HIPEAC Vision for Advanced Computing in Horizon 2020. pages 1–48, 2013.

[7] Magnus Halvorsen. Convolutional Neural Networks and their Potential Hardware Acceleration. Technical report, 2014.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE International Conference on Computer Vision*, 1998.

[9] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. 1969.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Buisness Media, LLC, 2006.

[11] DE Rumelhart, GE Hinton, and RJ Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[12] J. Leonard and M. A. Kramer. Improvement of the backpropagation algorithm for training neural networks. *Computers & Chemical Engineering*, 14(3):337–341, 1990.

[13] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, page 247, 2010.

[14] Xilinx. Zynq-7000 product table. http://www.xilinx.com/publications/prod_mktg/zynq7000/
7000-combined-product-table.pdf.

[15] Xilinx. Zyng-7000 overview. http://www.xilinx.com/support/documentation/data_sheets/ds1
Zynq-7000-Overview.pdf.

[16] Xilinx. AXI reference guide. http://www.xilinx.com/support/documentation/ip_documentati

[17] Kunihiko Fukushima. Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biol. Cybernetics*, 202:193–202, 1980.

[18] Kunihiko Fukushima and Sei Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. January 1982.

[19] PY Simard, Dave Steinkraus, and JC Platt. Best practices for convolutional neural networks applied to visual document analysis. *Seventh International Conference on Document Analysis and Recognition*, 2003.

[20] Dan C Cires, Ueli Meier, Jonathan Masci, and Luca M Gambardella. Flexible , High Performance Convolutional Neural Networks for Image Classification. *IJCAI'11 Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, pages 1237–1242, 2011.

[21] Dan Cirean, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural networks : the official journal of the International Neural Network Society*, 32:333–8, August 2012.

[22] D Ciresan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. *Advances in Neural Information Processing Systems*, (25), 2012.

[23] R.G. Girones and a.M. Salcedo. Implementation with FPGAs of a pipelined on-line backpropagation. *ICECS'99. Proceedings of ICECS '99. 6th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.99EX357)*, 2:8–11, 1999.

[24] K Benkrid and S Belkacemi. Design and Implementations of a 2D Convolution Core for Video Applications on Fpgas. *Digital and Computational Video, 2002. DCV 2002. Proceedings. Third International Workshopand Computational Video, 2002. DCV 2002. Proceedings. Third International Workshop*, (November):85–92, 2002.

[25] F. Cardells-Tormo and P. Molinet. Area-efficient 2-D shift-variant convolvers for FPGA-based digital image processing. *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, pages 209–213, 2005.

[26] Hui Zhang, Mingxin Xia, and Guangshu Hu. A multiwindow partial buffering scheme for FPGA-based 2-D convolvers. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(2):200–204, 2007.

[27] Antony W. Savich, Medhat Moussa, and Shawki Areibi. The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study. *IEEE Transactions on Neural Networks*, 18(1):240–252, 2007.

[28] Clement Farabet, Berin Martini, Polina Akselrod, Selcuk Talay, Yann LeCun, and Eugenio Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 257–260, May 2010.

[29] Vinayak Gokhale, Jognhoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops / s Mobile Coprocessor for Deep Neural Networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 682–687, 2014.

[30] Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, and Eugenio Culurciello. Memory Access Optimized Routing Scheme for Deep Networks on a Mobile Coprocessor. *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, 2014.

[31] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[32] Magnus Halvorsen. Convolutional Neural Network Accelerator, 2015. https://github.com/magnhalv/Skole/tree/master/Master_Project.

[33] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. MNIST Database, 1998. http://yann.lecun.com/exdb/mnist/.

[34] Taiga Nomi. Tiny-CNN, 2015.

[35] A Tisan, S Oniga, D MIC, and A Buchman. Digital Implementation of The Sigmoid Function for FPGA Circuits. *ACTA TECHNICA NAPOCENSIS*, 50(2):15–20, 2009.

[36] Jordan L Holt and Jenq-neng Hwang. Finite Precision Error Analysis of Neural Network Hardware Implementations. *IEEE Transactions on Computers*, 42(3):281–290, 1993.

[37] T Chen, Zidong Du, Ninghui Sun, Jia Wang, and Chengyong Wu. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[38] David Bishop. IEEE Purposed Fixed and Floating Point Packages, 2015. http://vhdl.org/fphdl/.

[39] Che W. Lin and Jeen Shing Wang. A digital circuit design of hyperbolic tangent sigmoid function for neural networks. *Proceedings - IEEE International Symposium on Circuits and Systems*, pages 856–859, 2008.

[40] ASUS. ASUS X550JK Specifications, 2015. https://www.asus.com/Notebooks_Ultrabooks/X550JK/specifications/.