A Configurable and Versatile Architecture for Low Power, Energy Efficient Hardware Acceleration of Convolutional Neural Networks

Steinar Thune Christensen

June 18, 2019

Abstract

Convolutional neural networks (CNNs) have become paramount in today's *Artificial Intelligence* (AI) and *Machine Learning* applications. This is true for image recognition in particular. This thesis presents a configurable, versatile and flexible architecture for hardware acceleration of CNNs that is based on storing and accumulating the entire feature maps in local memory inside the accelerator. This has been done while aiming to be able to process any type of CNN while consuming as low power as possible and achieving the highest possible energy efficiency, which refers to the number of operations per unit energy (measured in Multiply-Accumulate operations per unit energy, MACs/s/W or MACs/J). Several different versions of the architecture have been synthesized and tested using different configurations. It performs well when compared to the state-of-the-art, achieving an improved energy efficiency of over a factor 5 for select CNN layers. The most efficient configuration achieves 175 GMACs/s/W, while consuming 2.3 mW of power and occupying 585 KGEs (Kilo Gate Equivalents) of area at 1V supply voltage and a 100MHz clock. This is a significant improvement over Eyeriss [YuH17b] (a state-of-the-art accelerator) which has a maximal energy efficiency of 122.8 GMACs/s/W.

Sammendrag

Nevrale nettverk basert på *foldning* (CNNs) har blitt essensielle i dagens *Kunstig-Intelligens*- og *Maskinlærings*-anvendelser. Dette gjelder særlig bildegjenkjenning. Denne masteroppgaven presenterer en konfigurerbar, allsidig og fleksibel arkitektur for maskinvareakselerasjon av CNNs som er basert på å lagre og akkumulere hele *feature maps* i lokalt minne inne i akseleratoren. Dette har blitt gjort med et mål om å være i stand til å prosessere enhver type CNN med så lavt effektforbruk og så høy energieffektivitet som mulig. Energieffektivitet viser til antall operasjoner per energienhet (målt i antall multiplikasjon-akkumulasjon-operasjoner per energienhet, MACs/s/W eller MACs/J). Flere ulike utgaver av arkitekturen har blitt syntetisert og testet med ulike konfigurasjoner. Sammenliknet med dagens beste akseleratorer preseterer den godt, den oppnår en energieffektivitetøkning med faktor større enn 5 for utvalgte CNN-lag. Den mest energieffektive konfigurasjonen oppnår 175 MACs/s/W med et effektforbruk på 2.3 mW og et arealforbruk på 585 KGEs (Kilo Gate Equivalents) med 1V forsyningsspenning og en klokkefrekvens på 100 MHz. Dette er en betydelig forbedring over Eyeriss [YuH17b] (en av dagens beste akseleratorer) som har en maksimal energieffektivitet på 122.8 MACs/s/W.

Preface

This Master Thesis is a presentation of my work conducted during the spring of 2019. It is a continuation of a literature review project I conducted during the autumn of 2018 where I learned a lot about hardware implementations of *Artificial Intelligence* (AI).

The project is done for Nordic Semiconductor [Nor19] in Trondheim, Norway, and has been carried out under their supervision. I was asked to implement an AI accelerator in hardware. It proved to be quite a demanding path that I had put myself on. After many long days and weeks at the office, constantly grinding and thinking about what I should do next I managed to develop an architecture and achieve a result with which I am very satisfied. This work has made me a much better engineer and digital hardware designer.

Acknowledgement

I would like to express my sincerest gratitude to Nordic Semiconductor for giving me such an interesting task and to Omer Qadir for excellent technical and philosophical guidance. Thanks also to Sondre Nesset and Saeeid Oskuii for technical advice. I would like to thank my professor Snorre Aunet for guidance and especially for assistance on writing this thesis.

Further I would like to thank my good friend Jan Gulla for letting me use this wonderful LaTeX-template. Most importantly I would like to thank my dearest beloved Karine Avagian for an overwhelming amount of support and belief along with a fair amount of coercion when necessary.

Finally I would like to thank all my fellow students in the office A496 at Gløshaugen for excellent company throughout these months and the high quantity of low quality coffee.

Contents

List of figures

List of tables

Acronyms and abbreviations

NN	Neural network
ANN	Artificial neural network
CNN	Convolutional neural network
AI	Artificial intelligence
ReLU	Rectified linear unit
FM	Feature map
ifmap	input feature map of a CNN layer
ofmap	output feature map of a CNN layer
ASIC	Application-Specific Integrated Circuit
Ops	Operations
KGEs	Thousand/Kilo gate equivalents
MGEs	Million/Mega gate equivalents
MAC	Multiply-accumulate
PE	Processing element
IoT	Internet of things
1D	One-Dimensional
2D	Two-Dimensional
FSM	Finite state machine
MSB	Most significant bit
LSB	Least significant bit
conv layer	Convolutional layer of a CNN
RTL	Register-transfer level
MUX	Multiplexer
DUT	Device under test

Chapter 1

Introduction

Artificial intelligence (AI) is moving to the edge, meaning that not only high throughput, high power data centers perform it, but also low power mobile devices. More and more applications employ it, including autonomous vehicles, Internet of Things, medical equipment, mobile phones and other mobile devices, just to mention a few in a large and growing landscape. Normally, AI and machine learning algorithms are processed in software and often accelerated by a GPU, but the aforementioned applications require the development of specialized *Application-Specific Integrated Circuits* (ASICs), i.e. specialized hardware, in order to decrease power consumption and increase speed. In [Cat17], a huge thirty-year survey of the development of ASICs for Neural Networks Neuromorphic computing it is concluded that: "the need for a non-von Neumann architecture that is low-power, massively parallel, can perform in real time, and has the potential to train or learn in an on-line fashion is clear.". Most major companies in the electronics industry invest heavily in creating their own machine learning and neural network processors for instance Qualcomm's Snapdragon 855 [Qua18], Google's TPU [Nor18] and Intel's Loihi [Dav18] among others. However most of the work done by these companies is undisclosed and patented. Much work has also been done by academia, such as Eyeriss [YuH17b] from MIT and YodaNN [Ren16] from ETH Zürich, and this thesis draws a lot of inspiration from these.

Ever since AlexNet [Ale12] won the *ImageNet Large Scale Visual Recognition Challenge* [Li 18] in 2012, deep *convolutional neural networks* (CNNs) have been the state-of-the-art in image recognition, which is a common machine learning application. The goal of this thesis is to accelerate *convolutional layers* in CNNs while consuming as little power as possible. There are many different convolutional neural networks that perform very well in e.g. Image recognition (ResNet [Kai15], VGG16 [Kar14], AlexNet [Ale12] etc.), and new CNNs emerge and break records quite often. Until there is an established consensus on exactly which CNN is the very best and the most efficient, a hardware architecture for the processing of CNNs needs to be versatile and configurable, so that as many different CNNs as possible can be processed.

Contributions

This thesis presents an architecture for hardware acceleration of convolutional neural networks. The proposed architecture is versatile, flexible and configurable, aiming to be able to process any convolutional neural network. In this work only the convolutional layers are accelerated, not *pooling* nor the *fully connected* layers (theory in section 2.3). The reason for this is that, compared to the other layers, the convolutional layers typically have the longest computation time, as shown in [Luk15] and consume the most power as stated in [YuH18b]. The architecture has been compared with Eyeriss [YuH17b], a state-of-the-art accelerator for CNNs. The most important metric for comparison is energy efficiency, measured in operations per unit energy. The presented architecture is intended to be as easy to comprehend and intuitive as possible, which makes it differ somewhat from much of the literature.

This work also presents a hardware accelerator for a special type of convolutional neural network, namely *one dimensional* (1D) convolutional neural networks. These are used to recognize patterns in one dimensional signals that vary over time, like accelerometers, gyroscpoes and microphones. There exists much work on the applications of 1D CNNs, but the literature is scarse or non-existent on the hardware acceleration of such networks. Since these networks are reduced by one dimension compared to 2D, they are a lot smaller in size, thus in hardware they consume less power, occupy less area and have a lower computation time.

In this work only *inference* (theory in section 2.2.1) is performed. In [Cat17] it is stated that the algorithm of *backpropagation* (the algorithm with which most CNNs learn) is not typically thought of as an on-line method, meaning that it is not common to learn on-chip. So in this project the implemented CNNs are assumed to be pre-trained so no adjustments of weights nor biases will take place inside the accelerator. The well known pre-trained CNN VGG16 [Kar14] has been used for comparison. The performance has been compared to other state-of-the-art accelerators, with energy efficiency as the main metric for comparison.

Tools

The following tools have been used in this project.

- SystemVerilog used for implementation and verification of hardware.
- Questasim [Que19] version 10.7b used for simulation of hardware.
- Synopsys Design Compiler version J-2014.09 compiling 55nm CMOS [Syn19] for synthesis.
- Synopsys Spyglass version 2018.09-1 [Spy19] for power estimation.
- Python 3.6 [Pyt19] for software scripting and software implementations of the hardware accelerator. Matplotlib [Mat19] for plots.

- Keras [Ker19] with a Tensorflow [Ten19] backend. Used in Python to create neural networks in software for comparison with the presented hardware.
- Lucidchart [Luc19] to create figures.

Some terms from the SystemVerilog vocabulary will be used. Examples are *initial blocks*, meaning non-synthesizable code that is executed once, *always_ff* meaning sequential logic processes and *always_comb* blocks meaning combinatorial logic processes.

1.1 Structure of report

This thesis is organized as follows:

Chapter 2: Background. gives an extensive walkthrough of the theory behind neural networks. The focus is firstly on standard fully connected feed forward neural networks, and halfway through it switches to the theory behind convolutional neural networks. It is written so that also readers with little or no experience with neural networks should be able to follow. Those experienced within the subject might want to skim through or skip ahead. In the very end of the chapter there is a brief overview of a selection of previous relevant work that has been done within the subject of hardware acceleration of convolutional neural networks, including a literature review that was conducted as a preparation to this thesis.

Chapter 3: Accelerator for One-Dimensional Convolutional Layers. presents the accelerator for 1D convolutional neural networks. This chapter contains its own theory, implementation, results and discussion.

Chapter 4: Accelerator for Two-Dimensional Convolutional Layers. The architecture and the ideas presented in chapter 3 are further developed, from 1D to 2D. Hence all the well known 2D convolutional neural networks (e.g. AlexNet and VGG16) can be processed and used for comparison with other CNN hardware implementations. This chapter contains its own theory, implementation, results and discussion.

Chapter 5: Discussion. As the chapters 3 and 4 also contain their own more specific discussions, this chapter is a wrapup discussion on a higher level, aiming to discuss the broader points of the presented architectures.

Chapter 6: Conclusion. Concludes and summarizes main findings.

Chapter 2

Background

This chapter covers the relevant types of *neural networks* (NNs) that will be discussed in the chapters to come. All theory is taken from Michael Nielsen's online book [Mic15], unless otherwise is stated.

2.1 The artificial neuron model

The term *neuron* is borrowed from bioloogy referring to the neurons in the brain, however the *artificial* neuron is highly simplified in comparison. It takes any number of inputs a_i, multiplies them by their respective *weight* w_i, adds all of the products together with its own inherent bias b and the resulting value is used as an argument for an *activation function* (indicated by the f inside the circle of figure 2.1) that produces the final output activation value (see formula (2.1)). A neuron has only *one* output activation value a_{out}. The activation function can be any function at all, but typical ones are the sigmoid function (σ), the hyperbolic tangent (tanh) and the rectified linear unit ($ReLU$ [Sag17]). A graphical representation of a neuron is shown in figure 2.1.

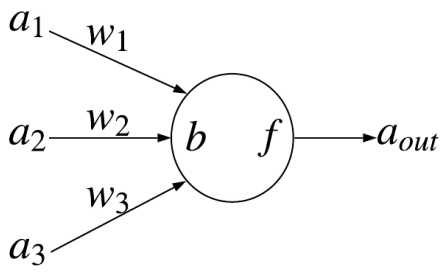


Figure 2.1: The basic artificial neuron model.

The neuron of figure 2.1 performs the calculation shown in formula (2.1).

$$a_{out} = f(b + \sum_{i=1}^{I} w_i a_i) \tag{2.1}$$

Where a_{out} is the output activation, f is the activation function, w_i is the weight of input i, a_i is the input activation of input number i, b is the neuron's inherent bias and I is the number of inputs. Figure 2.1 has a number of inputs $I = 3$.

2.2 Simple fully connected neural network

When several artifical neurons (shown in figure 2.1) are connected, they constitute a *neural network* (NN). A graphical representation is shown in figure 2.2.

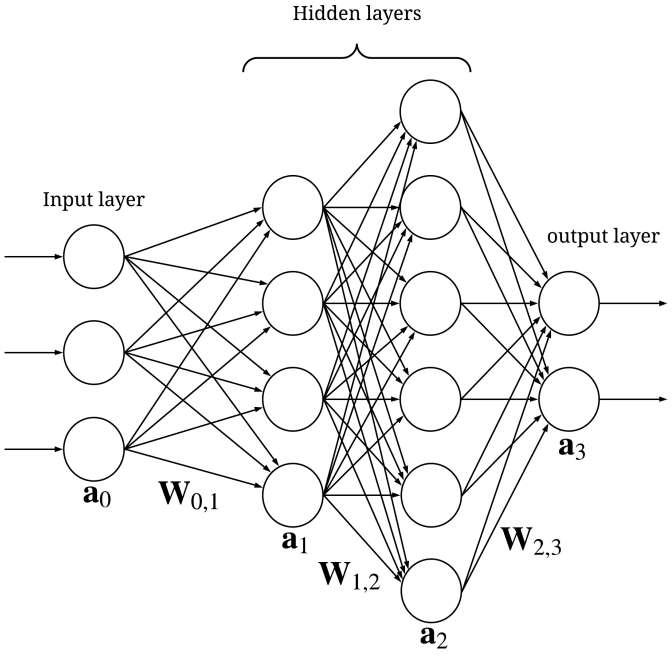


Figure 2.2: Simple fully connected neural network.

NNs are arranged in connected layers. When *all* neurons in a layer are connected to *all* neurons in the next layer, and this is the case for *all* layers, then the NN is said to be *fully connected*. A fully connected NN consists of one input layer, one output layer and any number of layers in between. The layers that reside in between are known as *hidden layers*. The weights are represented as matrices $\mathbf{W}_{n-1,n}$ going from layer $n-1$ to n. The output activations of one layer can be formulated in matrix-vector terms like shown in formula (2.2).

$$\mathbf{a}_n = f(\mathbf{b}_n + \mathbf{W}_{n-1,n} \times \mathbf{a}_{n-1}) \tag{2.2}$$

Where $\mathbf{a}_n$ is the vector of all output activations of layer n, $\mathbf{a}_{n-1}$ is the vector of all input activations of layer n, f is the activation function applied element-wise, $\mathbf{W}_{n-1,n}$ is the matrix of weights of connections mapping all neurons in layer $n-1$ to layer n, $\mathbf{b}_n$ is the vector of inherent biases for each neuron in layer n and $\times$ represents matrix multiplication. If the NN consists of more than one hidden layer then it is known as a *deep neural network* (DNN). It has been shown that deeper NNs are typically better than shallower NNs at classification tasks. It is generally accepted that deeper layers possess a more abstract representation of the input data than earlier layers, which allows the NN to extract more meaningful features. For example, in an image recognition task, one would not care so much about exactly which pixel is dark and which is bright. One would rather like to be able to see if a line is present and whether that line is generally curved or straight.

2.2.1 Training NNs

The phase known as *training* in an NN consists of adjusting the weights and biases until the network exhibits some predefined desired behaviour (e.g. image recognition). The most common way of doing this is by so-called *supervised learning*, which means that the desired output of the input data is known (i.e. input data is labeled). Then a *cost function* can be computed, which is a measure of how wrong the NNs output is. This function depends on all weights and biases of the network, which in a multilayer, fully connected network is a very high number of variables. Training consists of minimizing the cost function with respect to the weights and biases, this is commonly done using *gradient descent*, which includes calculating the gradient of each variable in the network. This is commonly done using the algorithm of *backpropagation* (chapter 2 of [Mic15] gives an excellent walkthrough), which includes computing the derivatives of activation functions, which means that having an easily differentiable activation function is hugely beneficial (e.g. sigmoid or tanh). During training it is common to test the network for several inputs subsequently, then compute the cost function of all of those inputs and then run backpropagation, this method is known as *batching*.

Overfitting should be avoided when training an NN. Just as when doing linear regression, you would rather have a straight line almost hitting all data points than a curved line hitting every data point exactly. This is because the former is more generalized and more likely to give good predictions. To avoid overfitting in NNs, techniques known as *regularization* (e.g. weight decay and dropout) are applied. These will not be discussed in detail here, but chapter 3 of [Mic15] gives a good description.

In general, a neural network's operation can be divided into 2 phases: the *training* phase, which consists of adjusting weights and biases and the *inference* phase which consists of using those weights and biases that were attained during training. Typically the training phase will consist of an NN looking at thousands of sets of data to *train* and optimize weights and biases. After training the NN can be used to do useful *inference*, where it would be used to classify unseen data, or some similar task. One final note about training is that deeper

networks are harder to train. This is due to the derivatives of the cost function with respect to the weights and biases. When these derivatives are calculated using backpropagation they are multiplied in long chains. These chains tend to either explode towards infinity or go to 0. These problems are known as the *exploding gradient problem* and the *vanishing gradient problem* respectively, and are one of the major problems with deep NNs.

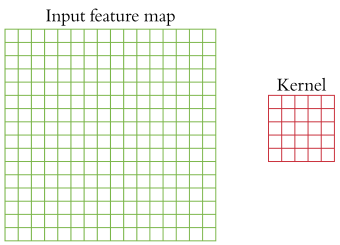
Training can also be done without labeled data, this is known as *unsupervised learning*. This can be done in many different ways. An example is self-organizing maps (SOMs) for clustering [Abh18] which is about maximizing certain neurons' responses to certain input data. *Reinforcement learning* is another way a neural network can learn, it is about acting in some environment, figuring out whether the action was desirable or not and adjusting weights accordingly (this is how AlphaGo [Dem17] works).

2.3 Convolutional neural networks

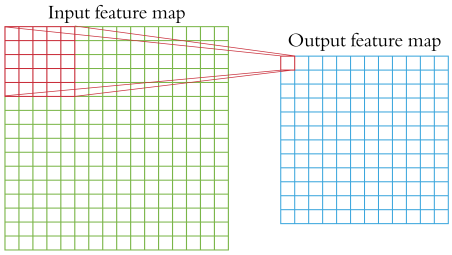
2.3.1 2D Convolutional neural networks

For image recognition the standard NNs have major weaknesses. Fully connected NNs have vast amounts of parameters, and they do not take into account the location of pixels relative to one another. Pixels on completely different ends of the image will be evaluated equivalently as pixels right next to each other, which typically is not useful. Pixels placed close to one another should have a greater impact on the image recognition result than those placed further apart. This is where *convolutional neural networks* (CNNs) are particularly useful. They make the computation a bit more complicated, but use fewer parameters and yield far better results.

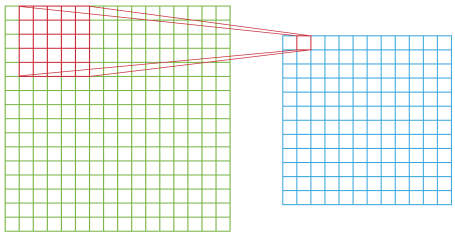
In a 2D CNN, instead of the weighted sum (formula (2.1)), a *2D convolution* operation is performed, as shown in formula (2.3). In a 2D convolution, a small set of weights arranged in a square 2D array (known as the *filter* or the *kernel*) is placed over a part of the input image. Then *one* output activation value for that section is calculated by taking the weighted sum of the pixels in the section and the weights in the kernel. The kernel is then shifted one pixel (or some other fixed number of pixels, this number is known as the *stride length*) and the same process, with the same kernel is repeated in all positions of the image until the end is reached. This takes the relative positions of pixels into account. A CNN is working in very much the same way as a regular NN, only that the neurons are replaced with pixels and the matrix multiplications of formula (2.2) are replaced with convolutions. Figure 2.3 illustrates a 2D convolution operation and how it can be perceived as a *sliding window*.



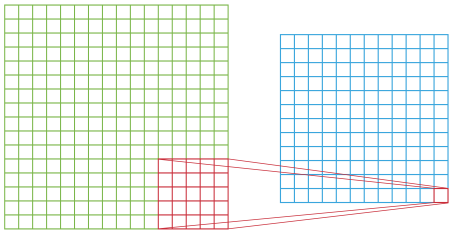
(a) 16×16 input feature map and 5×5 kernel



(b) Calculation of the first element in the output feature map, window position 1.



(c) Second element, window position 2.



(d) Last element and last window position.

Figure 2.3: Illustration of a single 2D convolution. The sliding window of a 2D convolution

The images processed by a CNN are typically called *feature maps*. That is because after doing convolutions the images are not containing anything concrete anymore, but rather they are representations of features in the original image. Figure 2.3 shows how each element in the output feature map is the sum of the element-wise multiplication of the kernel and the input feature map at the kernels current position. This is repeated for the whole input feature map until the entire output feature map is generated as shown in figure 2.3(d). The calculation of a single convolution is shown in formula (2.3).

$$o(x,y) = \sum_{a=0}^{b-1} \sum_{b=0}^{h-1} i(x+a, y+b) \cdot w(a,b) \tag{2.3}$$

Where $o(x,y)$ is the pixel at index (x,y) in the output feature map, $i(x,y)$ is the pixel at index (x,y) in the input feature map, $w(a,b)$ is the kernel value at index (a,b) in the kernel. b and h are the kernel width and height respectively. The output feature map is always going to be $b-1$ pixels narrower horizontally and $h-1$ pixels shorter vertically than the input feature map.

A CNN is arranged in layers just like a simple fully connected NN (presented in section 2.2), a graphical model is shown in figure 2.4.

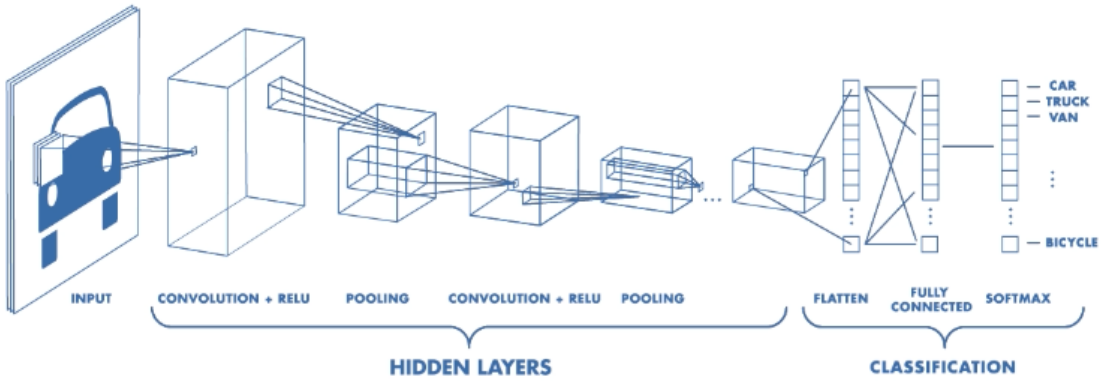


Figure 2.4: A graphical overview of a complete CNN. Figure taken from [Shy17]

Figure 2.4 shows the complete architecture of a CNN including convolutional layers, pooling layers and fully connected layers. In this work the main focus is on individual layers performing convolutions (*conv layers*). In that context some terms will be borrowed from *Eyeriss* [YuH17b] by Chen et al. The terms are *ifmap*, meaning an input feature map of a CNN layer, and *ofmap*, meaning an output feature map of a CNN layer. In one single convolutional layer there are, in general, several ifmaps and several ofmaps. Between each ifmap and ofmap there is one *unique* kernel that is used for the convolution.

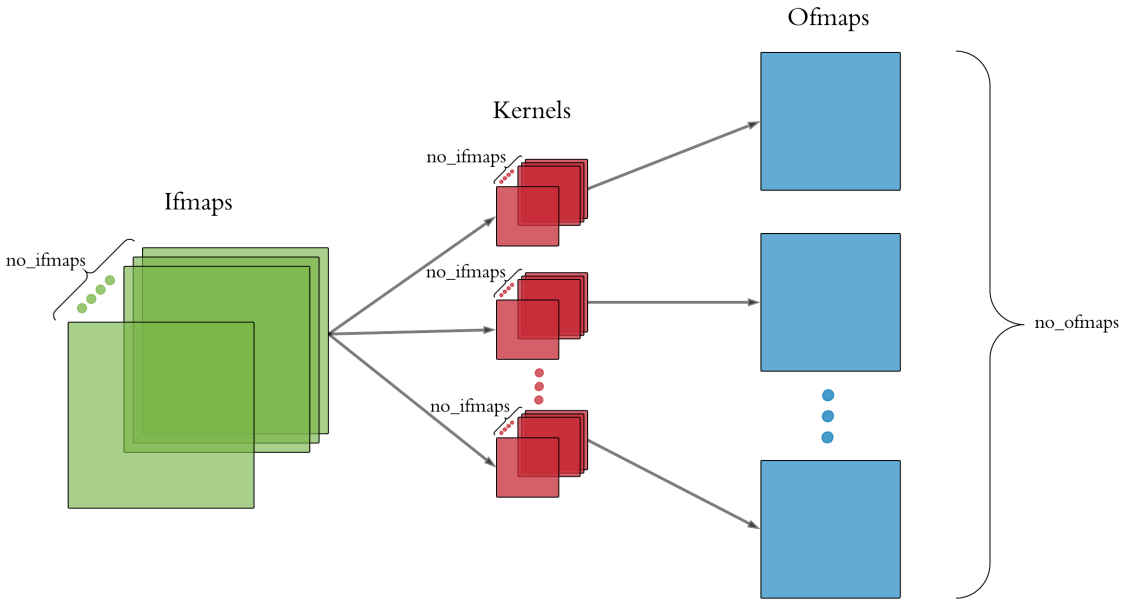


Figure 2.5: Ifmaps, kernels and ofmaps in one single convolutional layer

Figure 2.5 shows the relationship between ifmaps, kernels and ofmaps in a single CNN layer (see the appendix A.1 for an illustration of how the figure 2.5 fits into figure 2.4). A convolution is performed between each ifmap and its corresponding kernel in one of the

kernel groups. All of these convolution outputs are added together element-wise (plus some bias that is constant for that ofmap). The activation function is applied and then this produces one ofmap. The same procedure is repeated for all groups of kernels. Note the color coding used in this figure, green for ifmaps, red for kernels and blue for ofmaps. This color scheme will be used throughout this thesis.

Formula (2.4) from [Ren16] shows, in matrix form, the operation of a 2D convolution in a single CNN layer.

$$\mathbf{o}_m = \mathbf{C}_m + \sum_{n \in I} \mathbf{i}_n * \mathbf{w}_{n,m} \tag{2.4}$$

Where m denotes ofmap number, n denotes ifmap number, I represents the set of ifmaps, $\mathbf{o}_m$ is ofmap number m, $\mathbf{i}_n$ is input matrix number n, $\mathbf{C}_m$ is the bias to be added for ofmap number m, $*$ represents the convolution operator and $\mathbf{w}_{n,m}$ is the kernel that is used between ifmap n and ofmap m. This means that one ofmap is a sum of all ifmaps each of which have been convolved with its own kernel. To emphasize, for every ifmap, there is one unique kernel that maps it to every ofmap. Formula (2.5) is a form of the above formula (2.4) written out more explicitly.

$$o_m(x,y) = f\big(C_m + \sum_{n=1}^{I} \big(\sum_{a=0}^{h_k-1} \sum_{b=0}^{b_k-1} i_n(x+a, y+b) \cdot w_{n,m}(a,b)\big)\big) \tag{2.5}$$

Where m denotes ofmap number, n denotes ifmap number, I represents number of ifmaps, $o_m(x,y)$ is the value at index (x,y) for ofmap number m. i_n is ifmap number n, C_m is the bias to be added for ofmap number m, $w_{n,m}$ is the unique kernel that is used between ifmap n and ofmap m, h_k and b_k are the kernel height and width respectively. f is the activation function. In CNNs ReLU (shown in formula (2.6)) is normally the activation function.

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.6}$$

At the end of a CNN there are commonly one or more layers of fully connected NNs as shown in figure 2.2, these give the output of the CNN. It can be seen at the rightmost side of the CNN shown in figure 2.4.

Pooling

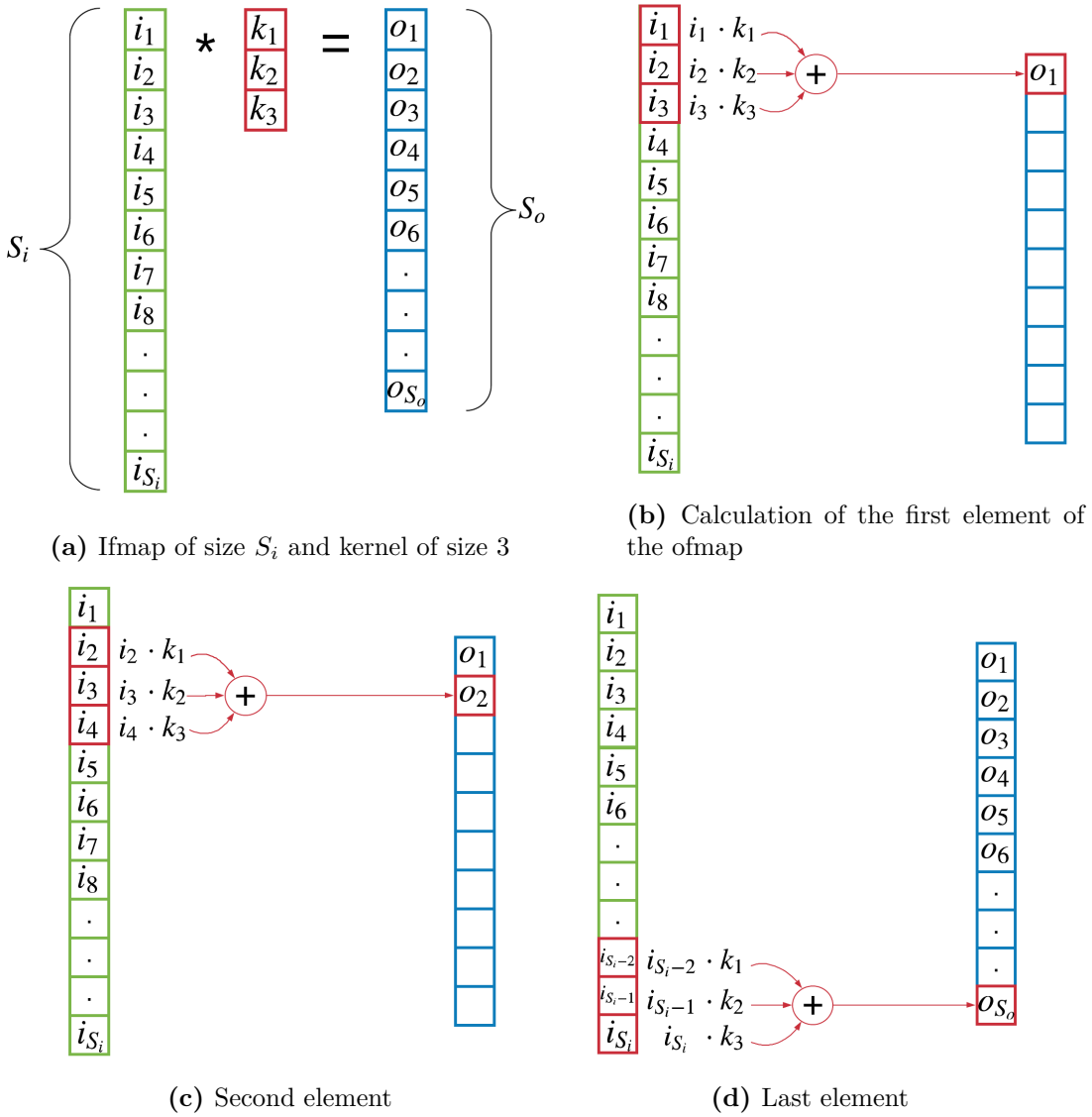
There are several more techniques that are commonly used in CNNs. Not all will be gone through in detail here, but a very common one is *pooling* (also known as subsampling). Pooling is a spatial downsampling of feature maps. The ifmap to a pooling operation is separated into non-overlapping sub-squares of a fixed size e.g. 2×2, and *one* output value

is calculated for each sub-square using a particular operation. The output value is placed in a corresponding position in the ofmap. Common pooling operations are average-pooling, average of all values in sub-square, and max-pooling, maximal value in sub-square. Sub-squares of size 2×2 produce outputs that are 4 times smaller in size than the inputs, 9 for 3×3 etc.

2.3.2 1D Convolutional neural networks

Chapter 3 of this thesis is devoted to *one-dimensional convolutional neural networks* (1D CNNs). These can be used for recognizing patterns in on dimensional data, like text or time series data such as sensor data. In [Xia16], [Min14], [Ste14] and [Soj16] several different applications are explored. One common applications is *Human activity recognition* (HAR), which involves using wearable accelerometers and gyroscopes to capture data and use 1D CNNs to recognize the behaviour of the wearers. In a 1D CNNs the feature map data elements are referred to as samples instead of pixels.

1D CNNs have a structure that follows 2D CNNs to a large extent, but feature maps and kernels are all one dimensional and the convolutions performed are 1D convolutions.



(a) Ifmap of size S_i and kernel of size 3

(b) Calculation of the first element of the ofmap

(c) Second element

(d) Last element

Figure 2.6: Illustration of a 1D convolution with a kernel of size 3. The sliding window of a 1D convolution

Figure 2.6 shows the computation of a 1D convolution. S_i is the number of elements in the ifmap and S_o is the number of elements in the ofmap. The number of elements in the ofmap is found by formula (2.7) where K is the kernel size.

$$S_o = S_i - K + 1 \tag{2.7}$$

A 1D CNN is arranged in layers of convolutions, pooling and fully connected layers just like 2D CNNs. Figure 2.4 shows the relationship between ifmaps, kernels and ofmaps in a single 1D CNN layer.

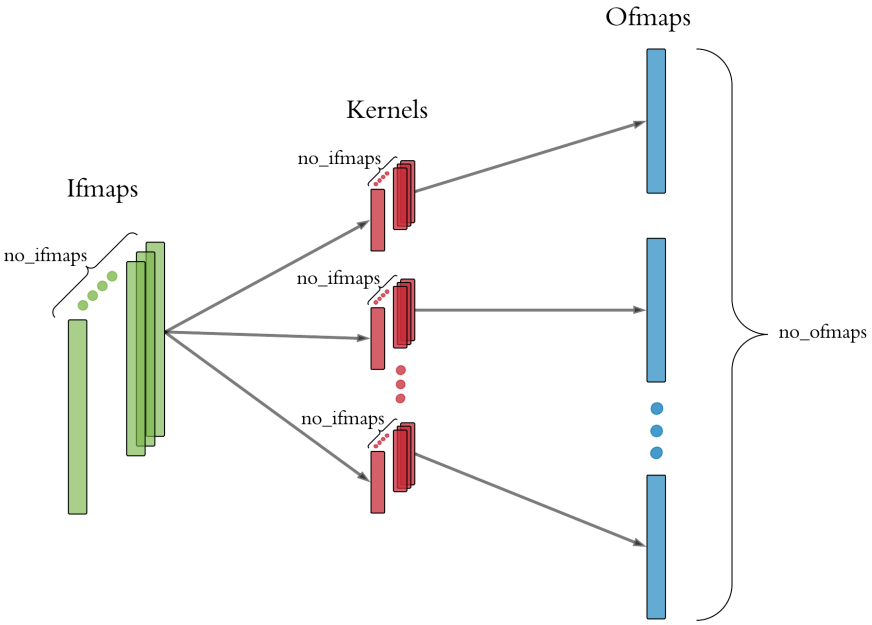


Figure 2.7: Ifmaps, kernels and ofmaps in a single 1D convolutional layer

The calculation of ofmaps is shown in formula (2.8).

$$o_m(x) = f(C_m + \sum_{n=1}^{I}(\sum_{a=0}^{b_k-1} i_n(x+a) \cdot w_{n,m}(a))) \tag{2.8}$$

Where m denotes ofmap number, n denotes ifmap number, I represents the number of ifmaps, $o_m(x)$ is the value at index (x) for ofmap number m, i_n is ifmap number n, C_m is the bias to be added for ofmap number m, $w_{n,m}$ is the unique kernel that is used between ifmap n and ofmap m, b_k is the kernel size and f is the activation function, which in CNNs is normally the ReLU.

2.4 Literature review

Prior to this thesis a literature review project [Chr18] was completed. This was done for investigation and exploration of the field of hardware acceleration of neural networks. In the project, four different hardware accelerators were investigated in detail, these were: YodaNN [Ren16], Eyeriss [YuH17b], EIE [Son16] and Hyperdrive [Ren18], all being accelerators for neural networks with a focus on low power consumption. With the exception of EIE, they all implement conv layers (section 2.3) each using its own set of optimizations. Table 2.1 shows the summary of the investigated architectures. MGEs are *Million Gate Equivalents* as a measure of area are refers to how many 2 input NAND gates of the same technology node would fit into given area. The architecture *EIE* is assumed to have 64 PEs. Ops denotes number of operations.

Table 2.1: Table comparing the architectures investigated in the literature review project.

Architecture	Energy Efficiency	Memory	Area	Technology
YodaNN	1 TOps/s/W	19.6 kB	1.9 mm^2/1.3 MGEs	65nm CMOS
Eyeriss	246 GOps/s/W	195 kB	12.3 mm^2/8.0 MGEs	65nm CMOS
EIE	3.8 TOps/s/W	10.3 MB	40.8 mm^2/40 MGEs	45nm CMOS
Hyperdrive	6.1 TOps/s/W	810 kB	1.92 mm^2/9.6 MGEs	GF 22nm FDX

The unit for energy efficiency Ops/s/W is equal to Ops/J which is the number of operations per unit energy. Operations however are not all equal in a CNN. Generally speaking, the only two operations performed in a CNN are multiplications and additions. Therefore this thesis will use the number of *Multiply-Accumulates* (MACs) instead of the generic "Ops".

Chapter 3

Accelerator for One-Dimensional Convolutional Layers

This chapter is a walkthrough of the implementation and design choices in the first CNN accelerator, which processes *one dimensional* convolutional layers. As mentioned in chapter 2, some terms will be borrowed from Eyeriss [YuH17b] by Chen et al. These are *ifmap* meaning an input feature map of a conv layer and *ofmap* meaning an output feature map of a conv layer. This is done for compactness.

3.1 Theoretical analysis

3.1.1 Considerations

When designing an accelerator for CNNs, there are a number of considerations that need to be taken. A selection of the main considerations is shown in the list below, they are further elaborated in the paragraphs that follow.

- Power consumption
- Energy Efficiency
- Speed
- Area
- Data reuse and movement
- Parallelization
- Number of input data passes
- Quantization and precision

- Functional flexibility

Power consumption

Power consumption is an important aspect of this thesis. Power is notoriously difficult to estimate because it is instantaneous and depends on which operation is being executed. If one were to execute operations very slowly, with low levels of parallelism, the power consumption would decrease, however the time taken to perform all operations would increase, hence the energy consumed for all operations would have accumulated over a large period of time. If speed and parallelism were to be increased, then power consumption would increase as well, but the time taken to execute all operations would be lower and the energy consumed could be more or less comparable to the first case. This leads to a trade-off between power consumption and throughput.

Energy Efficiency

The optimal point in the trade-off between power consumption and throughput lies where the energy used per *operation* is minimal, that is where the *Energy Efficiency* is maximal. Energy efficiency is typically measured in *Operations* per unit energy, however there are several different kinds of operations involved in the computation of a CNN, therefore the metric used in this thesis is *Multiply-Accumulates* (MACs) per unit energy (MACs/J or MACs/s/W). Energy efficiency will be measured using specific configurations of the presented accelerators implementing specific convolutional layers. Achieving a high energy efficiency is the main focus of this thesis.

Speed

An accelerator for a conv layer should to be sufficiently fast, no strict constraints are set here, but it is kept in mind. The throughput should to be as high as possible while still maintaining a low power consumption.

Area

The accelerators for conv layers discussed in section 2.4 have areas larger than 1 MGE. This project aims to produce an accelerator with a substantially smaller area than these. However as area decreases one would expect speed to decrease because of a lower potential for parallelism.

Data reuse and movement

In [YuH17b] and [YuH18a] there is a major focus on optimizing data movement and data reuse. As CNNs use a very large amount of data and weights, data movement can be more energy consuming than computation. The movement of data should be minimized and the reuse maximized.

Number of input data passes

Input data passes refers to how many times the CNN accelerator needs to be provided the same data. Ideally the accelerator is provided the same data only *once*, but as the internal storage capacity is inevitably limited, for a large conv layer the same data may have to be provided several times. This is the opposite of data reuse. Many conv layers are very large in size and number of parameters so this can make the number of data passes a great concern. Whenever some data is not reused maximally, then that data needs to be provided to the accelerator again. In the literature review project (section 2.4), it was discovered that the movement of data and needing to send data in to and out from the accelerator constantly is going to severely damage the energy efficiency. This was named the «I/O problem».

Parallelization

The reuse of data in a CNN also typically allows for parallelization. When designing any hardware accelerator a certain amount of parallelization is typically desired, this is to increase speed, although punishment comes in the form of extra area and power consumption (energy usage on the other hand might not be affected negatively). In the case of a conv layer there is a huge amount of parallelizable operations: e.g. a kernel is reused for all positions in its ifmap and the same ifmap is reused with many different kernels. In the case of batching, the same kernels are used for different ifmaps, although this thesis disregards this completely, as it is a technique for training, not for inference.

Quantization and precision

Quanization is of great importance in CNNs in HW. Most, if not all, software implementations of CNNs operate using floating point numbers. In low power hardware implementations however it is more common to use fixed point number representations because of its simplicity resulting in better power consumption, throughput and area. This does however, result in some lost accuracy, but has in the context of CNNs been shown to not necessarily affect the classification ability negatively, sometimes quite the contrary ([Son15], [M C16a]).

Many attempts have been made of quantizing weights heavily. Some even as far as having binary weights [Ren16], [M C16a], [M C16b]. In this project 16 bits have been used throughout, as it can give both a fair range and a fair precision. The accelerator aims to

be as versatile as possible, meaning that it should not be limited to only processing CNNs with binary weights. However no deep analysis has been done on how many bits are needed for accuracy, so the door is open to reduce the number of bits significantly. The weight data widths and feature map data widths have been parameterized with different parameters, allowing them to be adjusted independently.

Functional flexibility

Functional flexibility means that one would like to be able to process as many variations of CNNs as possible, with arbitrary sizes of feature maps, number of feature maps, sizes of kernels and number of layers. To support an arbitrary number of layers the accelerator should not be operating on one specific CNN with all its fixed parameters, but rather it should be a template in which any CNN could be implemented. To be able to have the highest functional flexibility possible it is a good idea to operate on *one* layer at a time. This means that no constraint will be put on the CNN depth, as one layer's outputs can be used as inputs of the next as many times as needed. All parameters inside one layer (kernel/ifmap sizes and number of ofmaps) are constrained to some respective maximal values. The accelerator supports all parameter sizes that are smaller or equal to the maximal values, making it flexible. Some methods to overcome the maximal values are presented in sections 4.3.1 and 4.3.2 so that CNNs larger than the maximal parameter values can be processed. However, these methods come at the expense of more data movement and result in sub-optimal data reuse.

3.1.2 Output stationary dataflow

The architecture presented here processes one layer at a time. It is provided with only one ifmap at a time and several kernels. The kernels that are provided are only those that are used in the convolutions using the current ifmap. This means that the provided ifmap and kernels can be used to compute partial sums of several ofmaps. Convolutions are then performed, and partial sums of all ofmaps are obtained in parallel. The partial ofmap sums are kept inside the accelerator while the next ifmap is provided, along with its corresponding kernels to perform more convolutions and accumulate on top of the previously computed ofmap partial sums. Figures 3.1 and 3.2 illustrate the first and second ifmap, respectively, being convolved with their corresponding kernels, which together give a partial sum of all ofmaps. The figures are modifications of the figure 2.7, where the color highlighting show which feature maps and kernels are currently being used. As the ofmaps are being kept and fully accumulated on chip in one location this is known as an *output stationary* dataflow architecture as presented in [YuH17a] and [Viv17].

Quoting [Viv17]:

> The output stationary dataflow is designed to minimize the energy consumption of reading and writing the partial sums [...]. It keeps the accumulation of partial

sums for the same output activation value local in the RF. In order to keep the accumulation of partial sums stationary in the RF, one common implemenation is to stream the input activations across the PE array and broadcast the weight to all PEs in the array.

Where RF means register file and refers to the memory where the ofmaps are stored. Partial sums refer to the ofmaps while they are being accumulated. PEs refer to *processing elements*, which are arranged in an array and will be presented in section 3.2. The architecture presented in this thesis applies an output stationary dataflow approach. The figures 3.1 and 3.2 show this approach used for computing ofmaps from ifmaps and kernels. The kernels are separated into groups, where each group corresponds to only one ofmap, but all ifmaps.

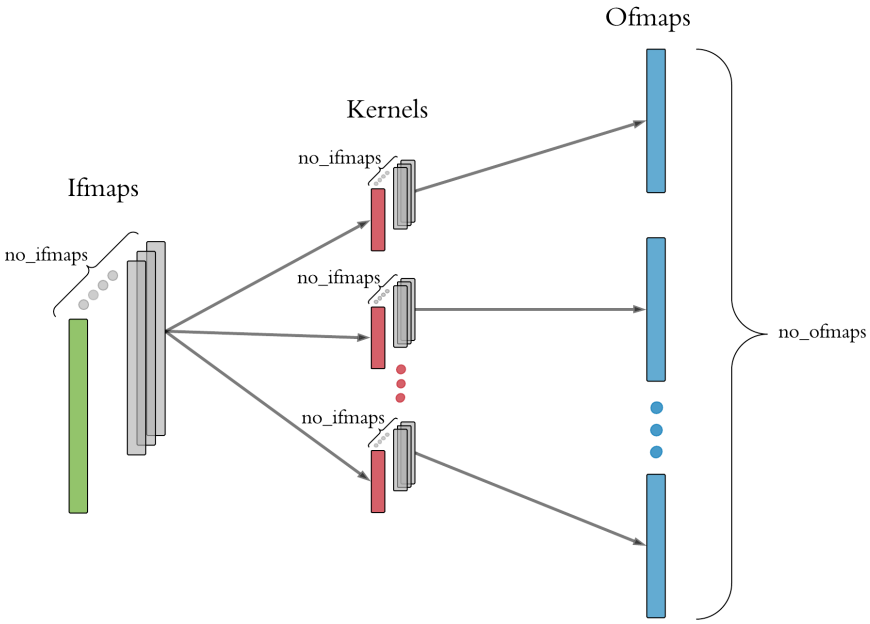


Figure 3.1: Processing of the first ifmap and its corresponding kernels producing partial sums of all ofmaps.

Formula (3.1) shows the calculation of the first partial sum of the ofmaps. Illustrated in figure 3.1.

$$psum_1(o_m(x)) = \sum_{a=0}^{b_m-1} i_1(x+a) \cdot w_{1,m}(a) \tag{3.1}$$

Where $psum_1(o_m(x))$ means the partial sum number 1 of ofmap m. The remaining variables are the same as in formula (2.8). The $psum_1()$ is not a proper mathematical function, it is merely an attempt to make it clear how the ofmap accumulation is happening.

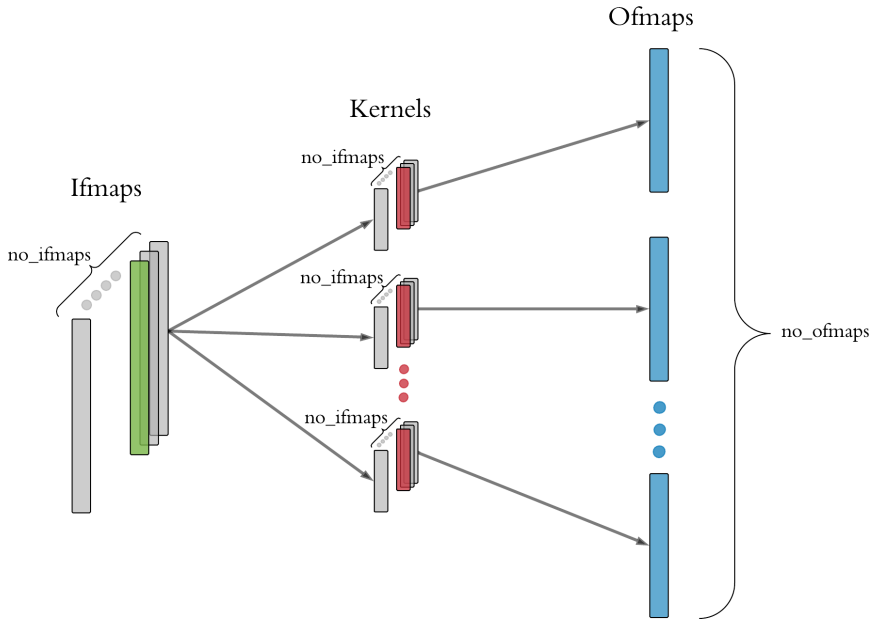


Figure 3.2: Processing of the second ifmap and its corresponding kernels adding on top of the partial ofmap sums produced in figure 3.1.

Formula (3.2) shows the calculation of the second partial sum of the ofmaps. Illustrated in figure 3.2. It adds on top of the first partial sum from calculation (3.1).

$$psum_2(o_m(x)) = psum_1(o_m(x)) + \sum_{a=0}^{b_m-1} i_2(x+a) \cdot w_{2,m}(a) \tag{3.2}$$

When this procedure is done for all n ifmaps, the bias C_m is added and the ReLU function is applied. Then it all adds up to formula (2.8), meaning that a full 1D CNN layer has been processed. The kernels can be kept stationary, while the one current ifmap "slides through" all of them simultaneously. Using this technique opens for parallelization of ofmap computation.

3.2 Hardware architecture

This section gives a walkthrough of the architecture designed for processing of 1D CNNs. The architecture will from here on be referred to as the *1D CNN accelerator*. Most accelerators for CNNs consist of an array of *processing elements* (PEs), each of which does some sort of MAC operation. This includes all architectures mentioned in section 2.4. The presented architecture also follows this structure because it is a scalable and configurable approach.

3.2.1 Top level

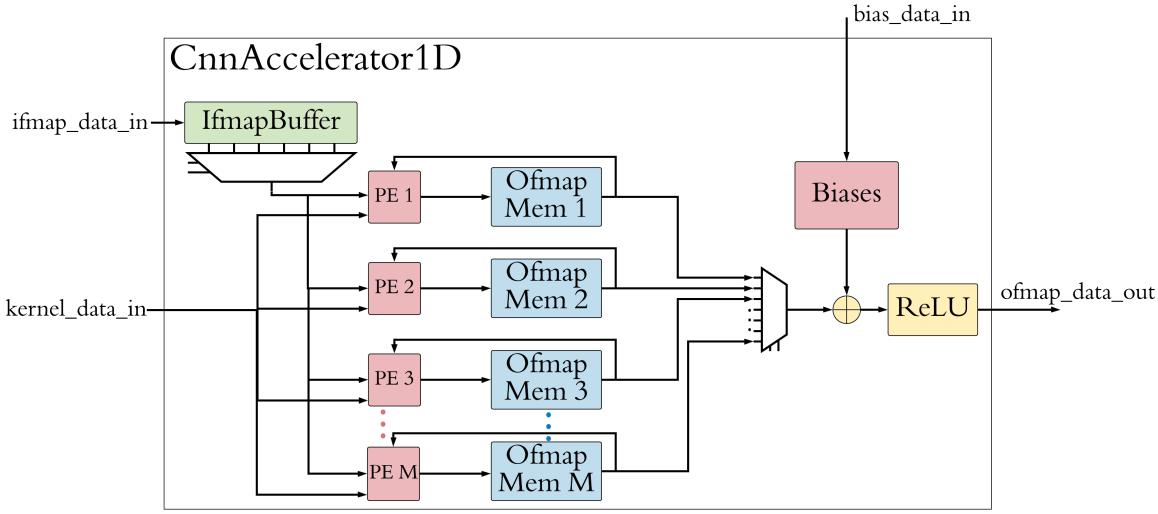


Figure 3.3: Architecture of the 1D CNN accelerator

Figure 3.3 shows the top level architecture of the 1D CNN accelerator. In the figure only data signals are depicted, the control signals are discussed in section 3.2.7. *IfmapBuffer* is a shift register containing a section of the ifmap. One of the samples stored in that shift register is multiplexed out to all PEs simultaneously. Each PE contains its own unique kernel that is used for a MAC operation with the ifmap value. When the PEs have finished processing, they have produced partial sums of single samples in the ofmaps, each PE corresponding to its own ofmap, this is then stored in its corresponding *OfmapMem*. This is done for all samples in the current ifmap. The OfmapMems are memory blocks that store the partial/intermediate ofmap results. The OfmapMems are fed back into the PEs so the current OfmapMem value is added to the output of the PE and stored in the same address of OfmapMem that was just read from. This serves as an accumulator. This means that the convolution of the current ifmap is added on top of the convolutions performed with the previous ifmaps. When all ifmaps have been streamed in and the ofmaps in OfmapMem are complete, then the ofmaps can be streamed out. *Biases* is a register containing n values which are the biases of the conv layer (C_m in formula (2.5)). There is one bias for each OfmapMem. The ReLU is the activation function. Data is represented as fixed point and in 2's complement.

Figure 3.3 has a clear correspondence to the figures 3.1 and 3.2. One ifmap is being convolved with m kernels, which corresponds to the IfmapBuffer and the m PEs. The convolutions produce m ofmaps, corresponding to the m OfmapMems.

Hardware parameters

To make the design as versatile as possible, there are several RTL hardware parameters that are configurable, but by the time synthesis is performed they need to be fixed. The parameters are as shown in table 3.1.

Table 3.1: Hardware parameters of the 1D CNN accelerator. Typ. range refers to the typical range of the parameter.

Name	Symbol	Typ. range	Description
`MAX_KERNEL_SIZE`	K	3 to 15	The maximal number of elements in a 1D kernel. Typically odd.
`MAX_NO_OFMAPS`	M	1 to 100	The maximal number of ofmaps that can be computed in parallel. Equal to the number of PEs.
`MAX_NO_SAMPLES`	S	10 to 200	The maximal number of samples in the ifmaps. Influences the size of OfmapMems.
`FM_DATA_WIDTH`	W_d	8 to 32	The number of bits in feature map data
`FM_FRAC_WIDTH`	-	0 to W_d-1	The number of bits in the fixed point representation's fractional part of the feature maps.
`WEIGHT_DATA_WIDTH`	W_w	1 to 32	The number of bits in weights. For the possibility to adjust only the weight precision independent of the feature map precision. See section 3.1.1 about quantization
`WEIGHT_FRAC_WIDTH`	-	0 to W_w-1	The number of bits in the fractional part of the weights.

Note that the maximal number of ifmaps is not specified as a parameter. This is because in principle any number of ifmaps can be used as input. For the data widths, W_d and W_w, 16 bits has been used throughout the work, while the fractional lengths have been varied depending on the CNN.

Inputs and outputs

All inputs and outputs of the accelerator are shown in table 3.2.

Table 3.2: Inputs and outputs of top level

Name	I/O	Width	Description
`rst`	in	1	Global active high reset.
`enable`	in	1	Global active high enable.
`ifmap_data_in`	in	W_d	Data signal. The ifmap data input.
`kernel_data_in`	in	W_w	Data signal. The kernel data input.
`bias_data_in`	in	W_d	Data signal. The bias data input.
`ofmap_data_out`	out	W_d	Data signal. The ofmap data output.
`new_layer`	in	1	Control signal that should be set high when beginning to process a new layer.
`layer_finished`	in	1	Control signal that should be set high when the last ifmap of a layer is being streamed in, valid ofmap data will be found in ofmap_data_out as a result.
`enable_ReLU`	in	1	Control signal that if high, the ReLU function will be applied to `ofmap_data_out`. If low the data bypasses the ReLU.
`no_samples`	in	$\lceil(log_2(S))\rceil$	Control signal. The number of samples in current ifmap. Must be lower than S.
`kernel_size`	in	$\lceil(log_2(K))\rceil$	Control signal. The size of the current kernel. Must be lower than K.
`no_ofmaps`	in	$\lceil(log_2(M))\rceil$	Control signal. The number of ofmaps, equal to the number of active PEs.
`valid_ifmap`	in	1	Streaming interface signal.
`valid_kernel`	in	1	Streaming interface signal.
`valid_bias`	in	1	Streaming interface signal.
`ready_ofmap`	in	1	Streaming interface signal.
`ready_ifmap`	out	1	Streaming interface signal.
`ready_kernel`	out	1	Streaming interface signal.
`ready_bias`	out	1	Streaming interface signal.
`valid_ofmap`	out	1	Streaming interface signal.

In table 3.2 log_2 is the logarithm with base 2. $\lceil x \rceil$ denotes rounding *up* to the nearest integer that is strictly greater than x. This applies also if x already is an integer, e.g. 3 bits are needed to represent the number 4 in binary although $log_2(4) = 2$ (essentially this is a $\lfloor x \rfloor + 1$ operation). Note that no_ifmaps is not an input. This is because only one ifmap is processed at a time and the accelerator does not require information about how many ifmaps are going to be used for computation, it is controlled by the `layer_finished` signal.

3.2.2 Ifmap Buffer

IfmapBuffer is a serial in parallel out shift register. The IfmapBuffer implements the *sliding window* over the ifmap data like shown in figure 2.6. The register is of size $K \times W_d$. The shifting of the IfmapBuffer is controlled by a signal `shift_enable`, which allows processing over several clock cycles in between each shift.

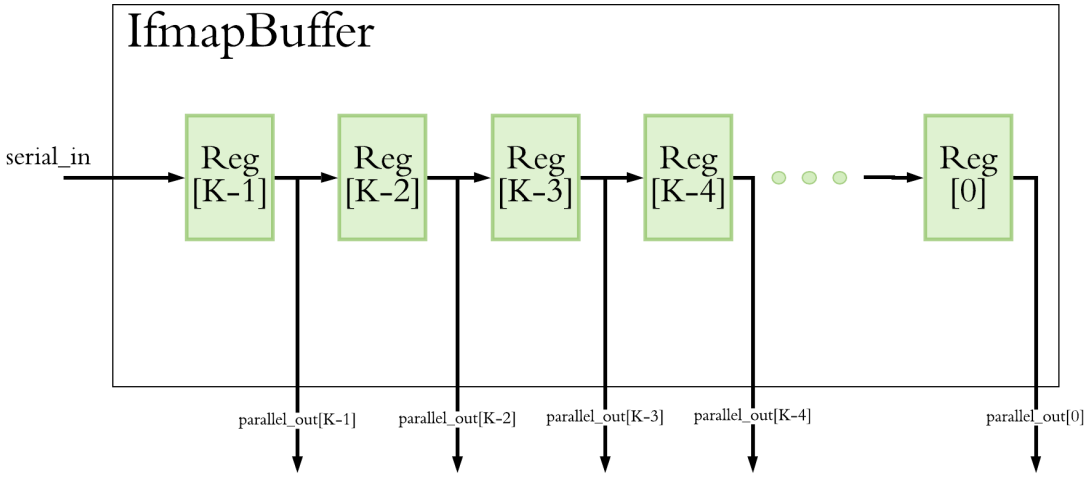


Figure 3.4: Architecture of the 1D Ifmap Buffer. K is the parameter from table 3.1.

3.2.3 Processing Elements

The processing elements are the heart of the accelerator and perform the MAC operations. Each PE contains one kernel stored in a register.

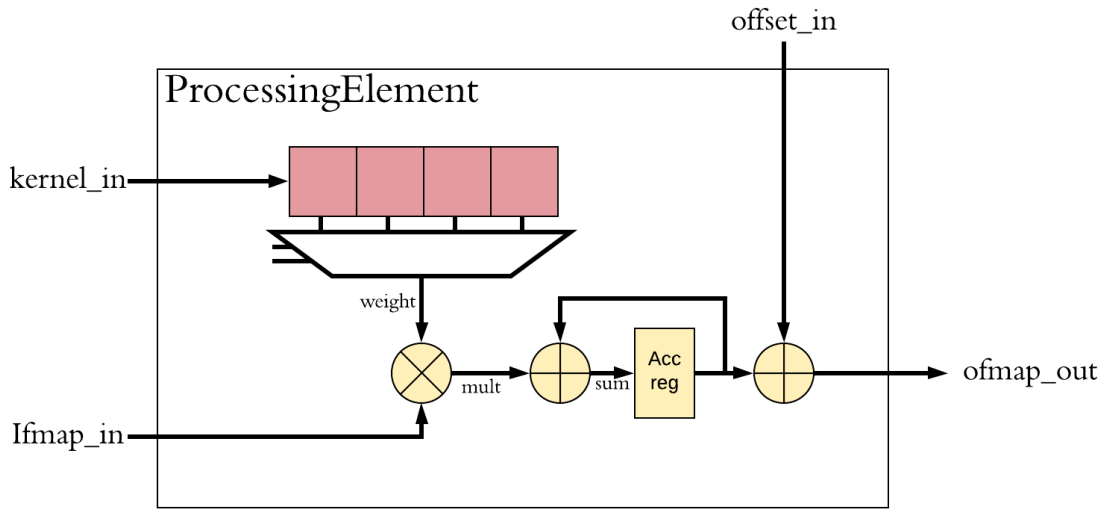


Figure 3.5: Architecture of a Processing Element with an example kernel register size (K) of 4.

The architecture of a PE is shown in figure 3.5. Only data signals are depicted in the figure. The kernels are stored in a register of size $K \times W_w$, indicated by the red boxes in the figure. Before processing, a phase of streaming in the kernel values takes place (e.g. k_1, k_2 and k_3 in figure 2.6(c)). During processing the `kernel_in` input is unused. When initializing the processing, the accumulator register *Acc reg* is reset to 0. An ifmap value is sent in through the `ifmap_in` input, it is multiplied by its corresponding kernel value from the kernel register. The product is sent into the adder and into *Acc reg* at the next rising clock edge. Then this is repeated for all ifmap values at the current kernel position (e.g. i_2, i_3 and i_4 in figure 2.6(c)). When all ifmap values have been processed, valid data is provided on `offset_in` and hence `ofmap_out` will contain a valid output. The product of multiplication in the PE is truncated, meaning that the bits less significant than the LSB will be omitted, resulting in a floor-type rounding operation.

Control signals

The control signals of the PE, all come from the main control path (section 3.2.7) and are as follows: 1) `kernel_sel` which is the selection signal of the multiplexer at the output of the kernel register. 2) `load_kernel` which if high, stores `kernel_in` in the kernel register. It should be low when processing. 3) `rst_acc` which resets the accumulator register. 4) `enable_acc` which enables the accumulator register.

Timing diagram

Figure 3.6 shows a timing diagram of a PE while processing.

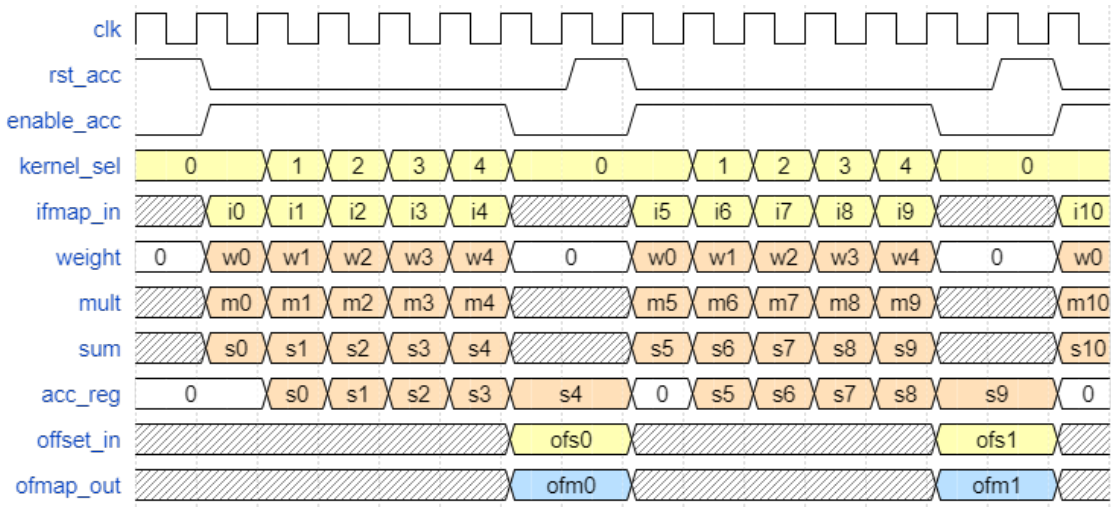


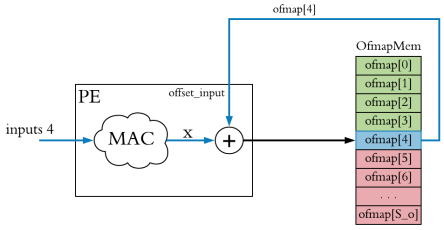
Figure 3.6: Timing diagram showing the operation of a PE with `kernel_size` = 5. It is assumed that the kernel reg has been loaded.

Where w0, s0, m0, ofm0 etc. are placeholder names given to weights, sums, products and ofmaps respectively, to give them unique identifiers. The timing diagram shows that the PE uses one extra clock cycle when resetting the accumulator. The reason for this will be explained in section 3.2.7.

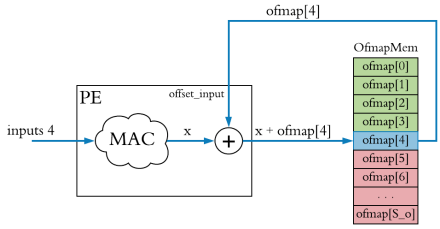
3.2.4 Ofmap memory

The OfmapMem blocks are memory blocks where the ofmaps values are stored and accumulated. To perform accumulation, a read operation has to be executed and followed by a write on the next clock cycle. In this case the OfmapMem blocks are implemented as flip-flops each of size $S \times W_d$, meaning that each OfmapMem has the storage space of the maximal ifmap size. They could be implemented as RAMs as only one address is required at a time in each memory block.

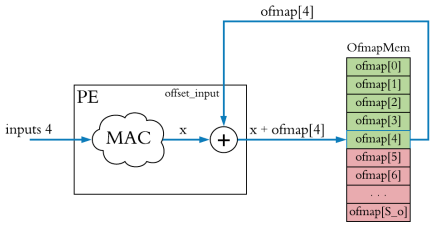
The sequence of operations that constitutes ofmap accumulation is shown in figure 3.7. The PE block in the figure disregards the intricacies of the PE discussed in section 3.2.3 and just names it a general MAC operation. This is for emphasizing that the final adder in the PE is what matters here.



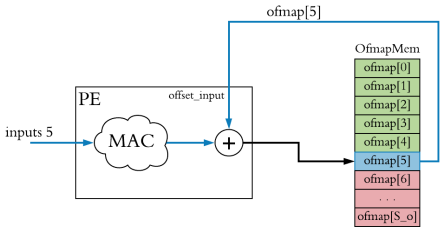
(a) Showing how an OfmapMem and a PE are connected. PE has calculated a value, which subsequently will be added to the OfmapMem value at current address, 4.



(b) The sum being calculated. Write will be performed on the coming positive clock edge.



(c) Positive clock edge has arrived and value is written to current address, 4.



(d) Address incremented to next position, 5. Where the same procedure will be executed.

Figure 3.7: The sequence of operations for performing accumulation in an OfmapMem block

At the output of the OfmapMem there is a MUX controlled by a signal named `ofmapMem_enable_acc` (not shown in figure 3.7). If this signal is 1, the MUX outputs the value stored at the current address in the OfmapMem. If it is 0 then the OfmapMem outputs a 0. This logic is essential for controlling when to accumulate in the OfmapMems, and when to rewrite.

3.2.5 Biases

The biases block is simply a register containing the biases of the ofmaps. It is loaded in the initialization phase. As a CNN has only one unique bias value per ofmap (C_k in formula (2.8)) the register has size $M \times W_d$. The register's output is chosen by the same signal that controls which ofmap is being streamed out.

3.2.6 ReLU

The ReLU activation function is described by formula (2.6). Data is represented in 2's complement form which means that the *Most Significant Bit* (MSB) can be used as an indicator of whether or not a number is negative. The ofmap data is streamed out one value at a time, so the ReLU block needs only one input and one output, each of width W_d. The architecture of the ReLU module is shown in figure 3.8.

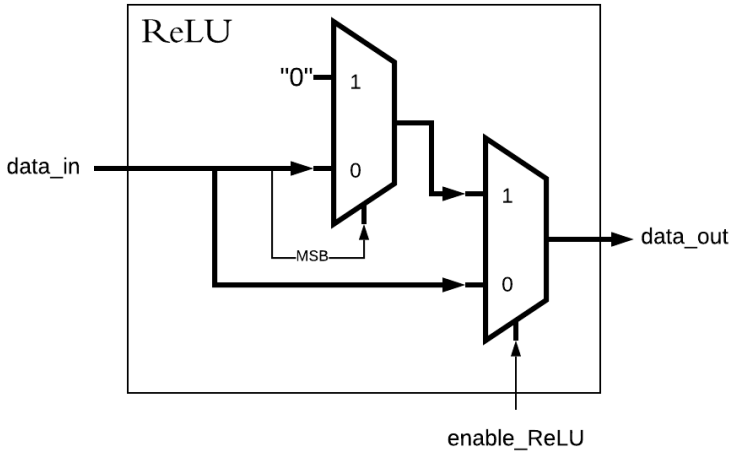


Figure 3.8: The ReLU module

3.2.7 Control path

The control path of the 1D CNN accelerator is organized as shown in figure 3.9. All sequential blocks (the FSM and Counters) run on the main clock.

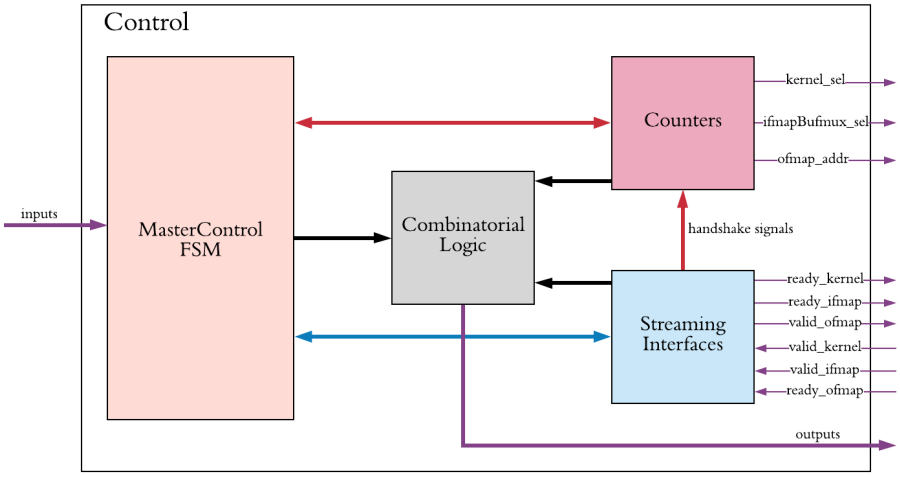


Figure 3.9: Architecture of the control block and a selection of inputs and outputs

The inputs and outputs that are shown in figure 3.9 are just a small selection of all signals, these are included for illustrational purposes. The control path is organized such that there is only one main state machine (MasterControlFSM) that keeps track of which operations are to be performed, and it does so by activating counters, input/output streaming interfaces and some combinatorial logic. Not all of the combinatorial logic will be gone through in detail.

Counters

Inside the *Counters* block in figure 3.9 there are 4 instances of a counter, they differ by their maximal count values which are as follows: `kernel_size`, `no_samples`, `no_ofmaps` and K. All counters run on the central common clock and are enabled by their own unique enable signals. The enable signals are set by either MasterControlFSM or the handshake signals from the *Streaming Interfaces* block. The usage of counters, with maximal count values controlled by signals is the main contributor of *functional flexibility* as described in section 3.1. It means that only by adjusting the input signals to the counters, the number of ofmaps, size of ifmaps and size of kernels can be adjusted.

Streaming interfaces

Inside the *Streaming interfaces* block in figure 3.9 there are four instances of a streaming interface that has been specially designed for this project. One instance for each type of data, that is, ifmaps, kernels, biases and ofmaps. The streaming interface blocks are very simple and fully combinatorial. They are written for being input data stream interfaces. Hence it sets an output *ready*, based on an enable signal from MasterControlFSM and takes an input *valid*, that is set by an external master which provides data. When both are high, a *handshake* signal is sent back to the MasterControlFSM. The handshake signals are mostly used to activate counters in the counter block, but in the case of the ifmap stream handshake signal, also to change states in MasterControlFSM. The streaming interface block is shown in figure 3.10.

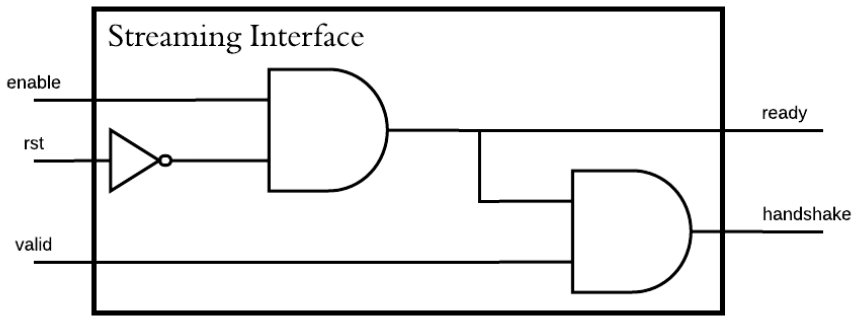


Figure 3.10: The streaming interface circuit

The streaming interfaces can be fully combinatorial as the MasterControlFSM is a Moore machine and the counters are fully synchronous. Then if the `valid` or `ready` signals were to switch spuriously in between clock cycles it would not affect the rest of the control path. For ofmap streaming, the `ready` and `valid` signals are simply switched around so the same architecture can be used as an output streaming interface instead of input.

MasterControlFSM

The state machine MasterControlFSM is described by the state diagram in figure 3.11. Its corresponding signals are shown in table 3.3. The FSM is a Moore machine so outputs are a function of state only. Hence outputs are written inside the states, whereas the inputs are shown along the edges that denote transitions.

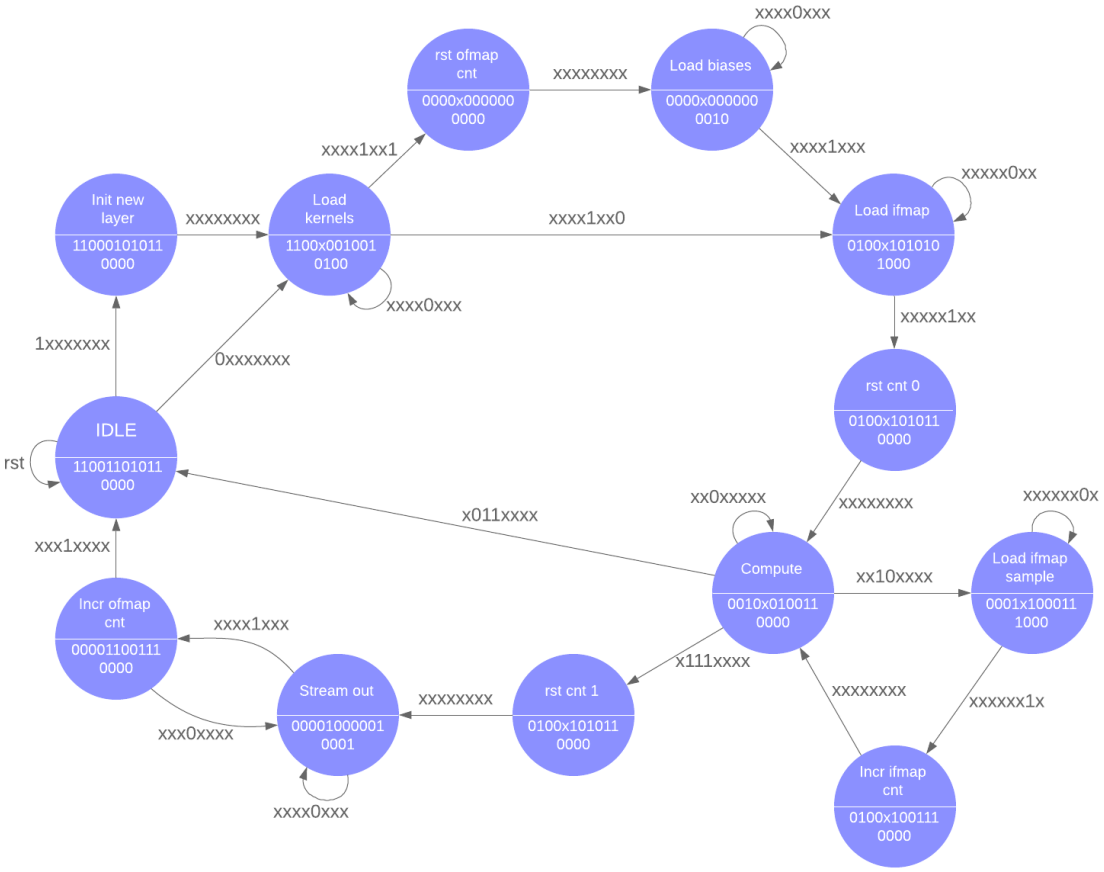


Figure 3.11: State diagram of the MasterControlFSM.

In table 3.3 the list of inputs and outputs of MasterControlFSM is shown, the ordering of signals is the same as depicted in the state diagram. Signal name disambiguation: `rst` means reset (local resets for counters or accumulators). `fin` means finished, it is a signal that is high for one clock cycle when a counter has reached its maximum value. `PE` refers to the processing elements, `no` refers to the "number of" something and `acc` refers to accumulators.

Table 3.3: inputs and outputs of FSM in figure 3.11

Inputs		Outputs	
#	**Signal name**	#	**Signal name**
1	`new_layer`	1	`ifmap_buffer_rst`
2	`layer_finished`	2	`PE_rst_acc`
3	`counterKernel_fin`	3	`PE_enable_acc`
4	`counter_ofmap_samples_fin`	4	`ofmapMem_write_enable`
5	`counter_no_ofmaps_fin`	5	`ofmapMem_enable_acc`
6	`counter_ifmapBuffer_fin`	6	`counter_kernel_rst`
7	`ifmap_stream_handshake`	7	`counter_kernel_enable`
8	`stream_biases`	8	`counter_ofmap_samples_rst`
		9	`counter_ofmap_samples_enable`
		10	`counter_no_ofmaps_rst`
		11	`counter_ifmapBuffer_rst`
		12	`ifmap_stream_enable`
		13	`kernel_stream_enable`
		14	`bias_stream_enable`
		15	`ofmap_stream_enable`

The idea behind the state machine is to perform operations in a loop. First load the kernels that are used from the current ifmap to all ofmaps (state: **Load kernels** in figure 3.11), then load the biases corresponding to all ofmaps (state: **Load biases**). Furthermore the ifmapBuffer is going to fill up with samples (state: **Load ifmap**) and then the accelerator is ready to process (state: **Compute**). While computing goes on there is a need for additional ifmap samples (state: **Load ifmap sample**), and when all ifmap samples have been processed, the FSM returns to **IDLE**. Then the same process is repeated for the next ifmap. When all ifmaps have been processed (indicated by the input `layer_finished`) then the ofmaps get streamed out (state: **Stream out**). The FSM has a synchronous `reset` signal and an `enable` signal. If `reset` is high it will automatically set **IDLE** as the next state and transition there on the next rising clock edge regardless of inputs. If `enable` is high, it will enable the state transitioning and if it is low, it will freeze the FSM in the state where it currently is.

The state **Incr ifmap cnt** is the reason for the extra clock cycle in the timing diagram of the PE, figure 3.6. In the state **Load ifmap sample** the FSM will wait for at least one clock cycle before receiving a new ifmap sample, and then exactly one clock cycle will be spent on incrementing the ifmap sample counter.

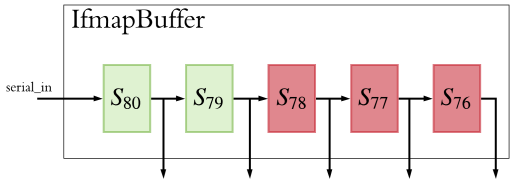
The OfmapMem enable acc signal

Note the "x" for output bit 5 (OfmapMem enable accumulator) in most states in figure 3.11. This is the only special output in this state machine. The OfmapMem enable accumulator signal is set to 1 by the IDLE state which means that the `offset_in` signal of the PEs (see figure 3.5) will be set to whatever is stored in OfmapMem and ofmap accumulation will take place. Then, if the `new_layer` signal is high before or exactly when the FSM starts off, the `ofmapMem_enable_acc` signal will be set to 0 (in the state **Init new layer**). This means that the `offset_in` signal of the PEs will be 0, meaning that the OfmapMems will be rewritten. The signal is set high again in the **Stream out** state, this is because the signal essentially is a read enable signal for the OfmapMems, and in the **Stream out** state the OfmapMems need to be read. This signal is not set by any other states. It is kept untouched during each cycle of the state machine. This results in it being stored in a latch. The latch is intentional. This is not ideal, but was used in the implementation, a possible alternative to this is mentioned in section 5.3.

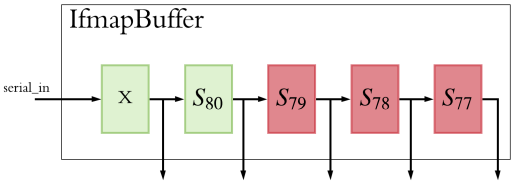
The Combinatorial Logic

The combinatorial logic block in figure 3.9 contains some additional combinatorial logic that needs to be applied to some select signals before sending them out to the top level.

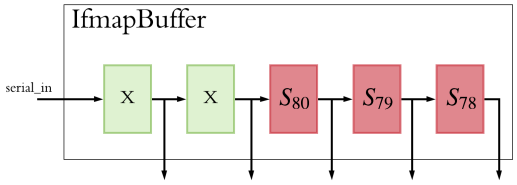
For instance the `ifmapBuffer_enable_shift` signal is controlling when the IfmapBuffer should shift. It is controlled mainly by the ifmap streaming interface circuit's handshake signal, but this handshake signal needs to be sent through some additional combinatorial logic to handle some special cases imposed by the following problem. One special signal in the combinatorial logic block is called `last_ifmap_shifts`. Its purpose is to ensure that all samples of the current ifmap get processed. When the signal `kernel_size`, which gives the current kernel size, is smaller than the maximal kernel size (K), the last samples in the ifmap will not be processed. They are stored in the IfmapBuffer, but not used for processing. The situation is illustrated in figure 3.12.



(a) When the last sample (number 80) has been shifted into IfmapBuffer.



(b) Performed additional shift to process sample S_{79}



(c) The last shift, so S_{80} can be processsed

Figure 3.12: Figures showing the last samples of an ifmap being shifted in IfmapBuffer. S_n denotes sample number n in the current ifmap.

Figure 3.12 shows a case where the number of samples in ifmaps are 80, `kernel_size` $= 3$ and $K = 5$ The registers marked in red are used, the green are not. The X's from figure 3.12 represent don't care, more specifically they are not valid ifmap data. To perform the shifts from 3.12(b) to 3.12(c) the additional logic is necessary because no additional ifmap data should be provided at the input. So the `last_ifmap_shifts` signal is set high after the last ifmap sample has arrived, and then the accelerator behaves as though there is valid ifmap data at the input although there is not. The `ready_ifmap` signal will in this case not be set high even though the IfmapBuffer performs shifts.

See the appendix section A.2 for a timing diagram showing how the calculation of an example 1D conv layer unfolds.

3.3 Verification

All subsystems described in section 3.2 have been tested, debugged and sanity checked individually. However the thorough testing has only been done on the top level. Figure 3.13 is a representation of how the functionality of the top level of the 1D CNN accelerator has been verified.

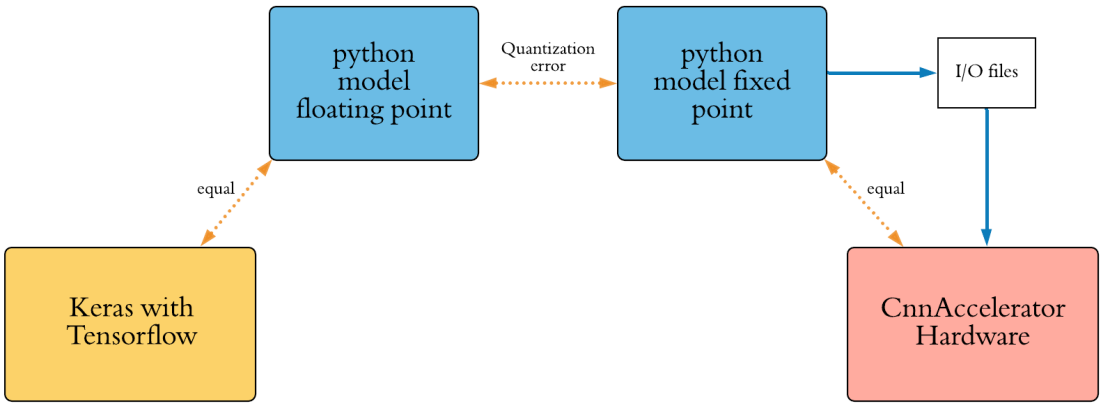


Figure 3.13: Functional verification method.

The idea behind the functional verification is to show that the accelerator will produce the correct result when the right data is provided. The procedure consists of the 4 following steps:

- The yellow box in figure 3.13 named *Keras with Tensorflow* is the producer of data, there, Keras with a Tensorflow backend (see the *tools* section chapter 1) is used to implement one single 1D CNN layer using the Keras function named *Conv1D*, and then write the kernels, biases, ifmaps and ofmaps to files. All data is written in double floating point precision.

- Verify that an ideal floating point precision version of the CNN accelerator produces the same result as in the ofmap file given the input data written to the kernel, bias and ifmap files. This is done in the box named *python model floating point* in figure 3.13. There will be some error, somewhere in the interval $[\sim 10^{-10}, \sim 10^{-3}]$ depending on how many values are accumulated, believed to be because of imperfections in the floating point number representation. This has been done by implementing a software version of the CNN accelerator in Python. The software mimics the functionality of the hardware, but as it is software, it can use floating points to show that it is equal to the Keras model.

- Run a script that implements software version of the CNN accelerator that is quantized in fixed point, done in the box named *python model fixed point* in figure 3.13. The only difference from this and the software model of the previous point is that the multiplications and additions are quantized in fixed point in the precision given in table 3.1. The quantized ofmaps are compared to those given by the ofmap files

produced by the Keras/Tensorflow model and the quantization error is identified. This script writes the ofmaps it obtains to a file so that it can be compared with the output of the hardware implementation. New quantized versions of the input data files are also written, so the hardware can use it as input.

- The red box named *CnnAccelerator Hardware* in figure 3.13 represents a testbench which instantiates the hardware CNN accelerator. The testbench uses the input files (kernels, biases, ifmaps) written by the fixed point python script as input, runs the hardware and obtains its ofmaps, these ofmaps are then compared with the ofmaps written by the fixed point script. If and only if all values are exactly equal will the test be considered a success.

3.3.1 Testbench

The testbench in the red box of figure 3.13 is shown in more detail in figure 3.14.

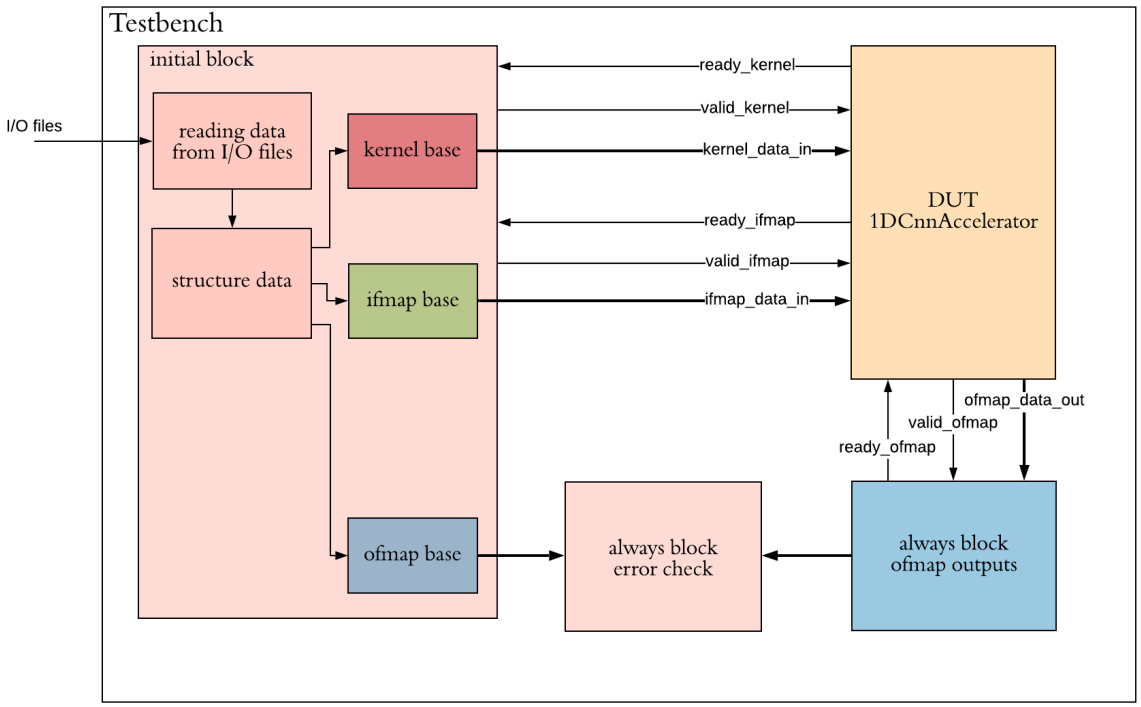


Figure 3.14: Structure of the testbench for the 1D CnnAccelerator. Biases have been omitted in figure as they are treated just like kernels. DUT means *Device under test*.

In this testbench there is one main initial block which operates using for loops to provide the right data when the HW accelerator sets its `ready` signals high, and capture the output data when it sets its `valid_ofmap` signal high. This requires the initial block to be structured to perform operations exactly in the order that the CNN accelerator's control path expects. This could have been made a bit easier using always blocks (done in section 4.3). In the *ofmap base* in figure 3.14 the expected ofmap values are stored. These values are the ones written by the software fixed point version in figure 3.13. In the *error check* always block

the ofmap values of the 1D CNN Accelerator are compared with the theoretically computed values of the ofmap base. Only when *all* values are equal will the test be considered a success. This testbench is developed for functional verification, it does not handle special corner cases to prove the robustness of the hardware, it strictly operates under ideal conditions.

Since no other hardware implementation of CNNs have been found there are no existing benchmarks to compare to. All that has been done to test the accelerator is to implement a 1D conv layer in Keras using the mentioned function, generate random ifmap data and see that they produce the same result as explained in the beginning of this section.

3.4 Results

A successful simulation of the 1D CNN accelerator has been run with the following parameters ($K = 15$, $M = 100$, $S = 80$) implementing a 1D conv layer with 5 ifmaps, an ifmap size of 80, kernel size of 7 and 20 ofmaps. The time taken is 46561 ns at 100 MHz and 1V supply voltage.

3.4.1 Syntheses

Area and power have both been estimated using Synopsys Design Compiler (see the Tools section in chapter 1). The results of some syntheses are shown in table 3.4.

Table 3.4: Results of synthesis with different parameter configurations.

M	**K**	**S**	**Area (KGEs)**	**Power (mW)**
1	15	80	17	0.3
32	15	80	451	3.5
32	30	80	521	6.7
32	15	200	949	3.6
100	15	80	1400	44

3.5 Discussion

3.5.1 Variations of the architecture

Here some variations of the CNN accelerator are discussed.

Weights in RAM

Using one central RAM to store all weights was one of the considered variations. However this was found to pose problems for the number of weights that could be fetched at one time, thus limiting parallelism. If several weights are concatenated into one address line, then some parallelism can be permitted, but RAMs do have quite restrictive maximal numbers of allowed bits per address line. For a RAM to be advantageous over a register to begin with, it needs to be of a certain size. If that storage can not be utilized to improve parallelism by having many active PEs simultaneously, then it becomes counter-productive to use a RAM. Therefore it was concluded not to use a RAM for weights.

Parallel Processing Elements

A type of PE that is parallel has also been suggested. The multiplications are happening in parallel and addition is happening in an adder tree. It is shown in figure 3.15.

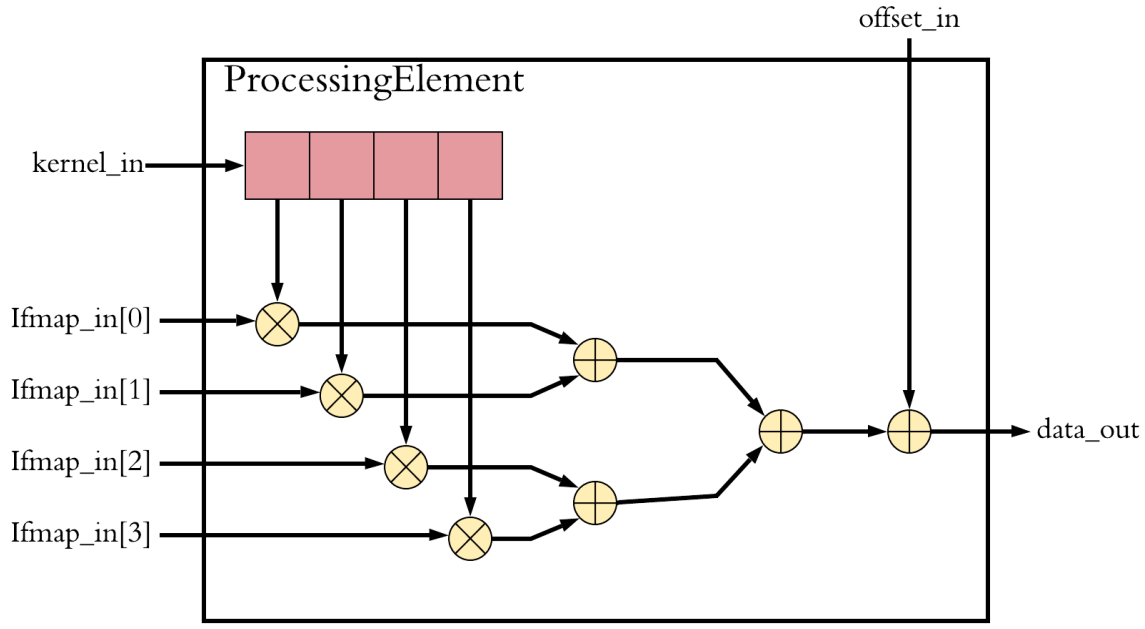


Figure 3.15: A parallel processing element with an example kernel size of 4

The PE shown in figure 3.15 would give a lot more speed (a factor of `kernel_size` in terms of clock cycles) and result in the PE producing an ofmap partial sum sample in a single clock

cycle. However it would damage the critical path significantly resulting in a lower maximal clock frequency. Segmentation (Pipelining) of the adder tree would improve the critical path and thus speed, but area would increase as well. The power consumption of a parallel PE would be a lot higher than the regular sequential PE (section 3.2.3), as multipliers are known to be power consuming operations (shown in [Son16]). For this reason the Parallel PE was not synthesized nor tested thoroughly.

OfmapMems as RAM(s)

The OfmapMems comprise quite a lot of storage space. It was briefly planned to create one central RAM in which to store the ofmaps. However this would affect parallelism like explained regarding putting the weights in RAMs. Therefore each ofmap could be implemented as its own RAM, but in the 1D CNN context each ofmap is often not very large (< 1000 samples per ofmap), therefore it is not obviously advantageous to use RAMs instead of registers. Implementing OfmapMems as several RAMs has been further elaborated in chapter 4.

3.5.2 Closing note

One final note is that the 1D CNN accelerator has not been developed and verified as much as planned and desired. In this thesis it serves mainly as an introduction to the chapter to come, about the 2D CNN accelerator, which has been further developed, verified and tested.

Chapter 4

Accelerator for Two-Dimensional Convolutional Layers

This chapter is a walkthrough of the implementation and design choices for the second CNN accelerator. This accelerator processes *two dimensional* convolutional CNN layers. It is based on the same architecure as presented in chapter 3.

4.1 Theoretical analysis

For designing the 2D CNN accelerator the same considerations as taken for the 1D CNN accelerator (section 3.1) need to be taken. One major difference is that the size of all data is greater. Thus more power consumption, more area and lower speed is to be expected. An illustration of the order of processing in this dataflow is described in the figures 4.1 and 4.2.

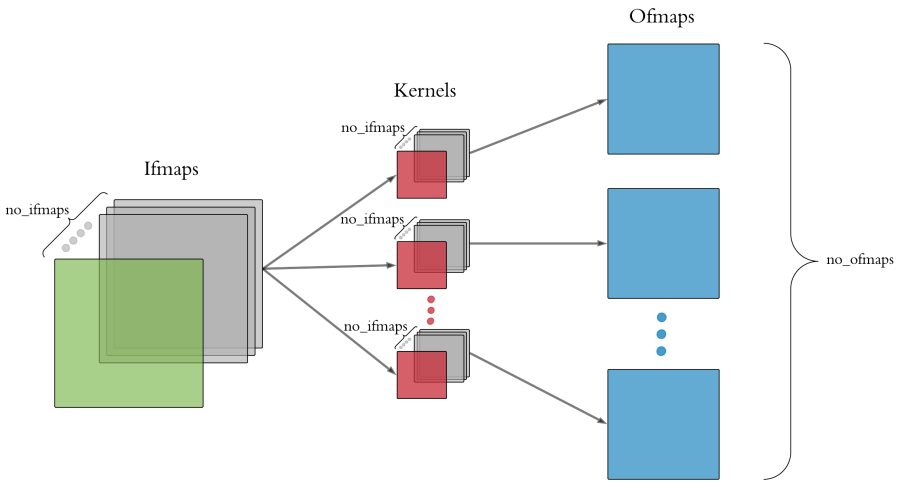


Figure 4.1: Processing of the first ifmap and its corresponding kernels producing partial sums of all ofmaps

Formula (4.1) shows the calculation of the first partial sum of the ofmaps. Illustrated in figure 4.1.

$$psum_1(o_m(x,y)) = \sum_{a=0}^{b_m-1} \sum_{b=0}^{h_m-1} i_1(x+a, y+b) \cdot w_{1,m}(a,b) \tag{4.1}$$

Where $psum_1(o_m(x,y))$ means the partial sum number 1 of ofmap m. The remaining variables are the same as in formula (2.5). The $psum_1()$ is not a proper mathematical function, it is merely an attempt to make it clear how the ofmap accumulation is happening.

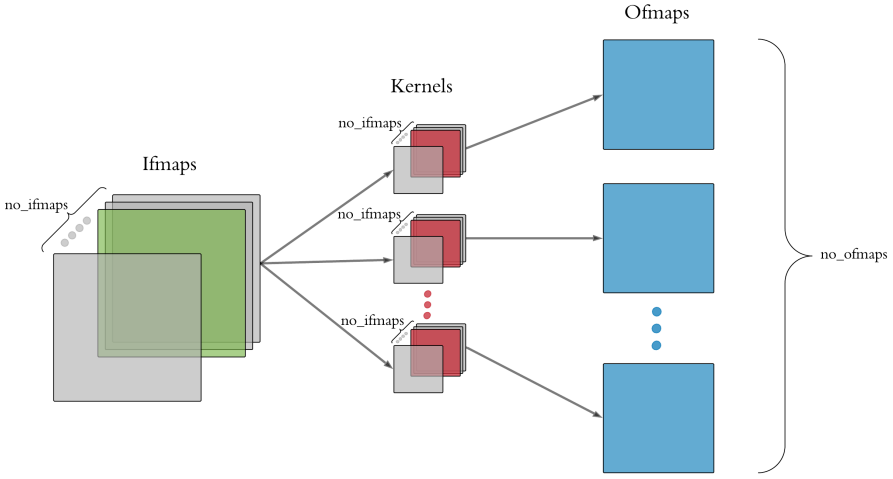


Figure 4.2: Processing of the second ifmap and its corresponding kernels adding on top of the partial ofmap sums produced in figure 4.1

Formula (4.2) shows the calculation of the second partial sum of the ofmaps. Illustrated in figure 4.2. It adds on top of the first partial sum from calculation (4.1).

$$psum_2(o_m(x,y)) = psum_1(o_m(x,y)) + \sum_{a=0}^{b_m-1} \sum_{b=0}^{h_m-1} i_2(x+a, y+b) \cdot w_{2,m}(a,b) \tag{4.2}$$

When this procedure is done for all n ifmaps, the bias C_m is added, the ReLU function applied and it adds up to formula (2.5).

4.2 Hardware architecture

This section presents the hardware architecture of the accelerator for two dimensional convolutional layers, from here on referred to as the *2D CNN accelerator*. It is very similar to the 1D CNN accelerator from chapter 3, in the top level diagram of figure 4.3 there are no structural differences. Square kernels are assumed and the OfmapMems are implemented as SRAMs.

Top level

The top level of the 2D CNN accelerator is shown in figure 4.3.

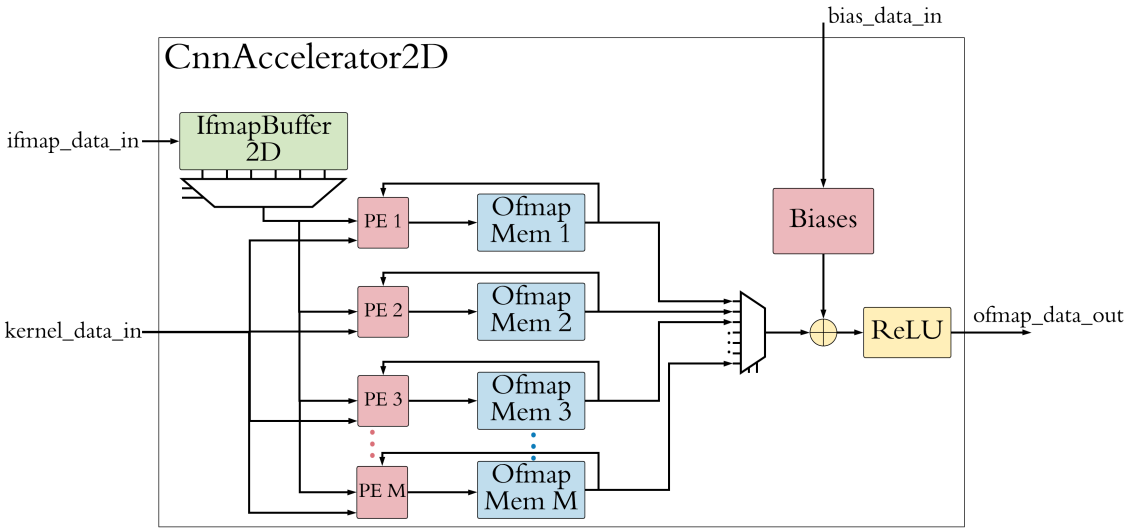


Figure 4.3: Architecture of the 2D CNN accelerator top level

Hardware parameters

The SystemVerilog hardware parameters of the design are shown in table 4.1.

Table 4.1: Hardware parameters of the 2D CNN accelerator

Name	Symbol	Typ. Range	Description
MAX_KERNEL_WIDTH	K	1 to 15	The maximal number of weights is one row of a 2D kernel. Square kernels are assumed, thus only one size identifier is required. The max kernel size will be K^2
MAX_NO_OFMAPS	M	1 to 100	The maximal number of ofmaps that can be computed in parallel
MAX_IFMAP_WIDTH	B	10 to 100	The maximal number of pixels in one ifmap row.
FM_DATA_WIDTH	W_d	8 to 32	The number of bits in feature map data
FM_FRAC_WIDTH	-	0 to W_d-1	The number of bits in the fixed point representation's fractional part of the feature maps.
WEIGHT_DATA_WIDTH	W_w	1 to 32	The number of bits in weight data. For the possibility to adjust only the weight precision independent of the feature map precision.
WEIGHT_FRAC_WIDTH	-	0 to W_w-1	The number of bits in the fractional part of the weights.

Another choice that has to be made, although not specified as a SystemVerilog code parameter is the size (number of words) of the OfmapMems. This is an independent variable and it influences the choice of B. It is set in the implementation of the OfmapMems themselves. More on this in section 4.2.3.

Inputs and outputs

All inputs and outputs of the accelerator are shown in table 4.2.

Table 4.2: Inputs and outputs of top level

Name	I/O	Width (bits)	Description
rst	in	1	Global active high reset.
enable	in	1	Global active high enable.
ifmap_data_in	in	W_d	Data signal. The ifmap data input.
kernel_data_in	in	W_w	Data signal. The kernel data input.
bias_data_in	in	W_d	Data signal. The bias data input.
ofmap_data_out	out	W_d	Data signal. The ofmap data output.
new_layer	in	1	Control signal that should be set high when beginning to process a new layer.
layer_finished	in	1	Control signal that should be set high when the last ifmap of a layer is being streamed in, valid ofmap data will be found in ofmap_data_out as a result.
enable_ReLU	in	1	Control signal that if high. The ReLU function will be applied to ofmap_data_out. If low the data bypasses the ReLU.
ifmap_width	in	$\lceil(log_2(B))\rceil$	Control signal. The width of the current ifmap.
ifmap_height	in	16	Control signal. The height of the current ifmap.
kernel_width	in	$\lceil(log_2(K))\rceil$	Control signal. The width of the current kernel. Square kernels are assumed.
no_ofmaps	in	$\lceil(log_2(M))\rceil$	Control signal. The number of ofmaps, equal to the number of active PEs.
valid_ifmap	in	1	Streaming interface signal.
valid_kernel	in	1	Streaming interface signal.
valid_bias	in	1	Streaming interface signal.
ready_ofmap	in	1	Streaming interface signal.
ready_ifmap	out	1	Streaming interface signal.
ready_kernel	out	1	Streaming interface signal.
ready_bias	out	1	Streaming interface signal.
valid_ofmap	out	1	Streaming interface signal.

4.2.1 Ifmap Buffer 2D

Distribution of data is the biggest difference between 1D and 2D convolutions and most of the distribution of data takes place in the IfmapBuffer. Therefore the IfmapBuffer is what differs the most between the 1D accelerator and the 2D accelerator, it is also the most complex part of the system. This section explains how the 2D IfmapBuffer is designed.

A simplified version

A 2D convolution is processed using the *sliding window* technique shown in figure 2.3. As the window slides across an ifmap, all pixels, except along the borders, are going to be used more than once. This is a type of data reuse (section 3.1). Hence it is advantageous to load all pixels once, keep them in memory as long as they are needed and throw them out as soon as they cease to be.

The basic idea behind the 2D IfmapBuffer using a shift register to shift in the ifmap pixels row-wise. The shift register is split into a new row for every full ifmap row. The register should be able to contain full ifmap rows. This results in a constraint on the ifmap width, hence the parameter B. The number of rows in the shift register does not need to be greater than the maximal kernel height, kernels are assumed to be square, so the number of rows in the ifmapBuffer is K.

A simplified view of the IfmapBuffer with some example parameters is shown in figure 4.4.

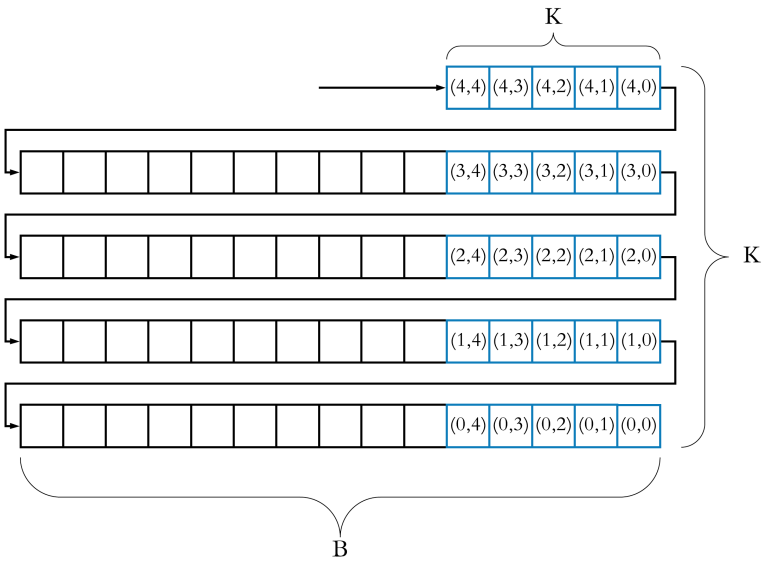


Figure 4.4: A simplified view of the architecture of IfmapBuffer with $B = 15$ and $K = 5$

All of the registers marked in blue in figure 4.4 are parallel outputs. They represent the

current position of the sliding window. The unmarked registers contain the pixels that are not currently used, but will be.

Configurable ifmap width and kernel width

When the IfmapBuffer is organized like shown in figure 4.4, only an ifmap width equal to B can be processed. To support different ifmap widths the architecture of figure 4.5 is used. The idea is that the value stored in the last register of each row can be loaded into *any* register in the next row. The architecture is drawn with a smaller B for simplicity.

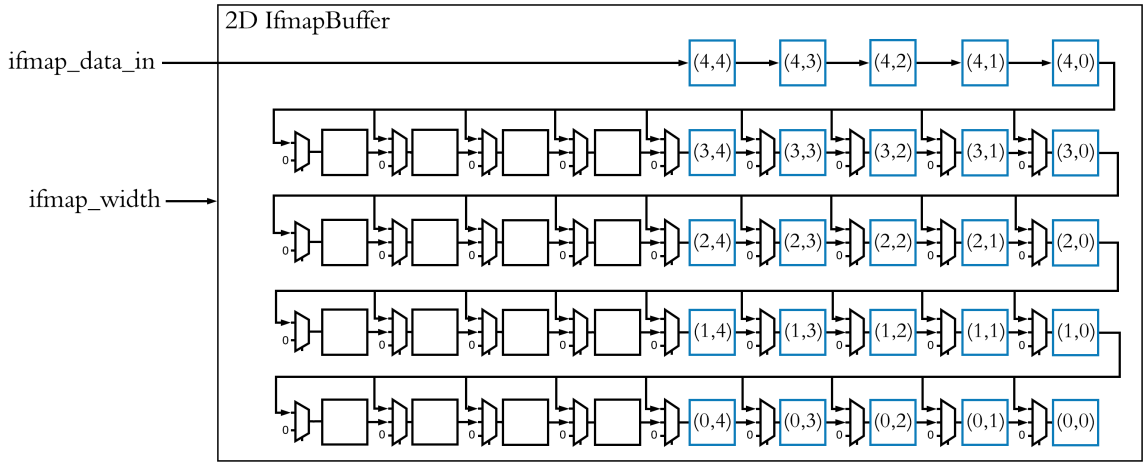


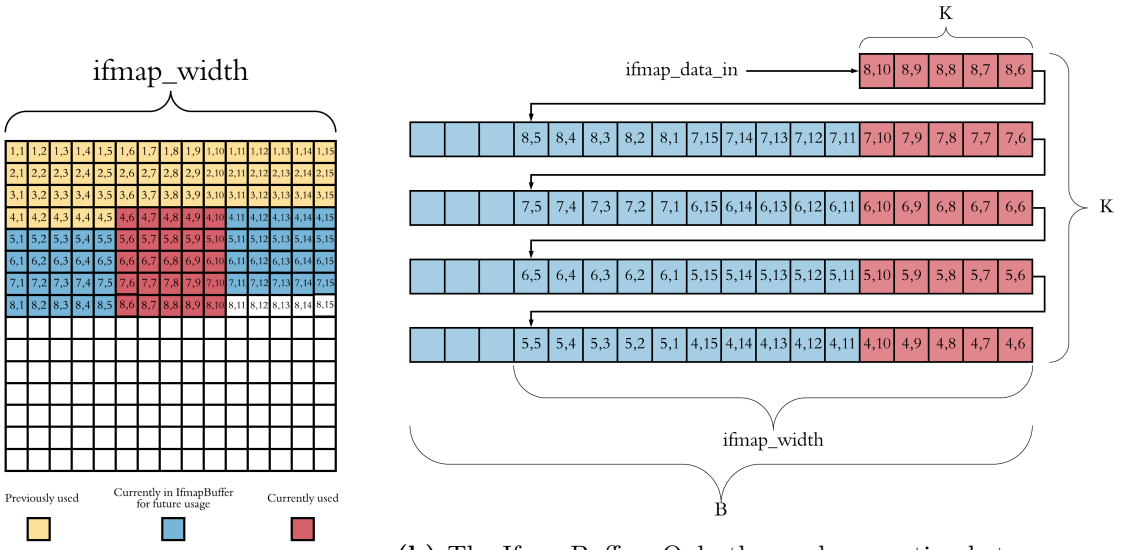
Figure 4.5: IfmapBuffer with configurable width, using multiplexers for configurability. $K = 5$ and $B = 9$.

The ifmap width then needs to be an input, its usage is not shown in figure 4.5, but it controls all multiplexers choosing into which register to load the value from the previous row. *One* register in each row is loaded with the value from the last register in the row above. The registers to its *right* act as regular shift registers, while the ones to the *left* are unused and loaded with zeros. In this manner all ifmap widths that are smaller than or equal to B can be processed. All rows are controlled in the same way by the ifmap width input.

The size of the kernel can be controlled outside of the IfmapBuffer. As all of the marked registers are parallel outputs, one can outside utilize only some of the outputs, thus effectively all kernel widths smaller than K can be processed.

The sliding window

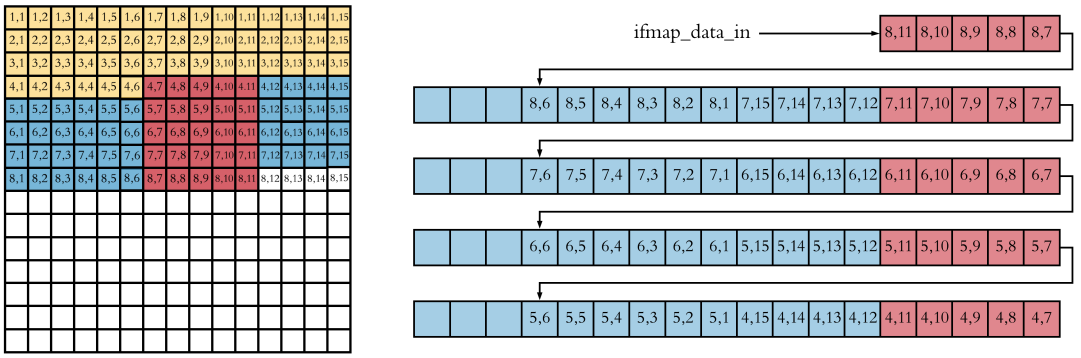
How the sliding window technique is achieved is shown visually in the figures 4.6 and 4.7.



(a) The ifmap.

(b) The IfmapBuffer. Only the used connection between rows is shown.

Figure 4.6: Showing the correspondence between the ifmap and the IfmapBuffer. `ifmap_width` = 15, $B = 18$ and $K = 5$.



(a) The ifmap.

(b) The IfmapBuffer.

Figure 4.7: The next stride position of the sliding window

4.2.2 Processing elements

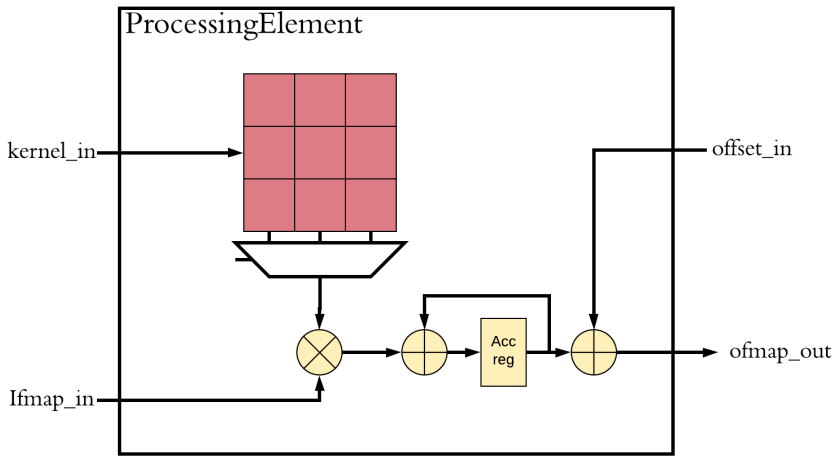


Figure 4.8: Architecture of the 2D Processing Element with an example kernel register of size 3^2.

Figure 4.8 depicts only data signals and not control signals. The main difference between this PE and the PE from section 3.2.3 is that the kernel stored in the *kernel reg* (the red boxes in figure 4.8) is of size $K^2 \times W_w$. Another difference is that the multiplication-accumulation is using a round half-up instead of truncation. A round half-up is the round operation where in the case where the distance up and down are equal, it always chooses to round up, this gives a slight positive bias that could be noticeable when lots of values are accumulated. Resulting in a significantly smaller quantization error when converting from floating point numbers to fixed point numbers (section 4.3).

The timing diagram in figure 3.6 is descriptive also of the 2D Processing element, the only differences are a larger 2D kernel register and two address identifiers for it, one for row and one for column.

4.2.3 OfmapMems

The OfmapMems are functionally equal to the ones described in section 3.2.4, but in this case, the ofmapMems are implemented as SRAMs. The reason is that when the size of a memory exceeds a certain limit a RAM is advantageous over a register in terms of area and power. The RAMs are compiled using MobileSemiconductor's compiler SP-HSLV-TS55EF_V1.43 which compiles an SRAM using TSMC 55nm low power CMOS process [Mob19].

The RAMs need to have a fixed size upon compilation, simulation and synthesis. That puts constraints on the size of the ofmaps that can be stored in each of them, which consequently puts constraints on B. The maximal size that has been used is 8192 words and 16 bits per word. 16 bits per word is because the 16 bit feature map data is what has been used the most in this project. 8192 was the maximal number of words that the compiler tolerated.

If square ofmaps are assumed (which might not be the case), then the maximal width of the ofmaps will be $\lfloor\sqrt{8192}\rfloor = 90$. Therefore a logical value for B is 90. One could have $B > 90$ and add the constraint that ifmaps need to be wider than they are tall, however this has not been considered. There are ways to overcome the limitation on the maximal ifmap width (discussed in section 4.3.1). Formula (4.3) shows the value of B generically.

$$B = \lfloor\sqrt{N_w}\rfloor \tag{4.3}$$

Where N_w is the number of words of width W_d in the ofmapMems. Formula (4.3) is only a suggestion to what B should be. The convolutions make the ofmaps `kernel_size - 1` smaller than ifmaps, so the ifmaps do allow a slightly larger size than given in formula (4.3), but that depends on `kernel_size`. $B = 90$ is used here as a default for OfmapMems as RAMs of size 8192.

4.2.4 Control path

The control path of the 2D CNN accelerator is more complex than that of the 1D CNN accelerator. So the main FSM has been divided into several smaller FSMs, while maintaining one central FSM that controls all the others. This results in the main FSM being a lot simpler and able to keep track of only high level operations like when to stream and compute, instead of low level operations like resetting specific counters.

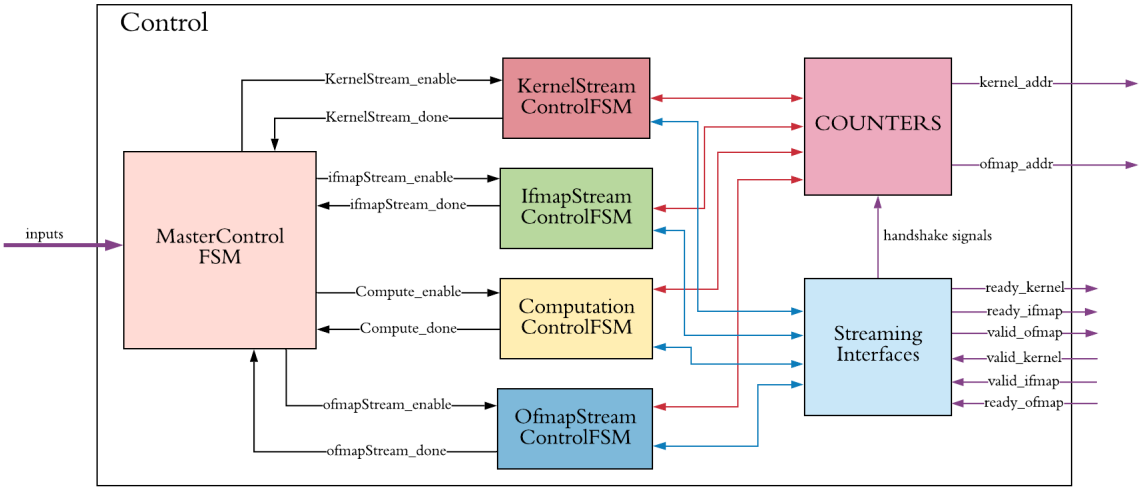


Figure 4.9: The architecture of the control path of the 2D CNN Accelerator. This also contains a *Combinatorial Logic* block as figure 3.9, but is not included for illustrational purposes.

The FSMs are described in the following paragraphs. The streaming interfaces are exactly the same as described in section 3.2.7.

Counters

Inside the *Counters* block in figure 4.9 there are 8 instances of a counter, they differ by their maximal count values which are as follows: 2 counting to `kernel_width` (one for rows and one for columns), `no_ofmaps`, size of used ifmapBuffer, number of pixels in ifmap, `ofmap_width`, `ofmap_height` and `ofmapAddr` (the current address used in the Ofmap-Mems). The counter that counts to `ofmapAddr` is enabled every time either the counter to `ofmap_width` or `ofmap_height` are enabled, hence no multiplier nor adder is needed to get the ofmap address based on the ofmap width and height.

MasterControl FSM

All of the following FSM described on the pages to follow have `reset` and `enable` like the masterControlFSM described in section 3.2.7.

The state diagram of the main central state machine MasterControlFSM is shown in figure 4.10, the inputs and outputs of the FSM are presented in table 4.3 which follows.

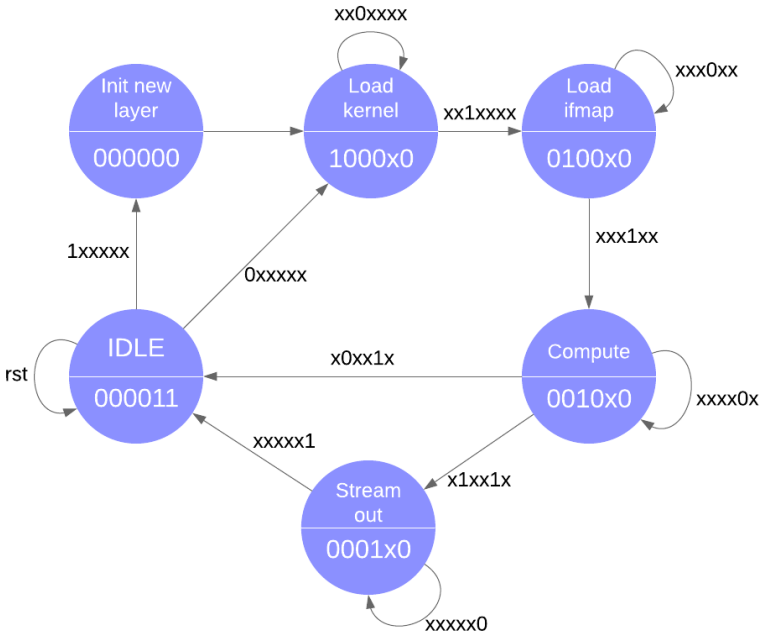


Figure 4.10: State diagram of the MasterControl FSM

The x's in the state diagram in figure 4.10 represent that the signal of that position is not set, resulting in a latch like explained in section 3.2.7.

Table 4.3: 2D MasterControl FSM inputs and outputs of FSM in figure 4.10

Inputs		Outputs	
#	**Signal name**	#	**Signal name**
1	`new_layer`	1	`kernel_stream_FSM_enable`
2	`layer_finished`	2	`ifmap_stream_FSM_enable`
3	`kernel_stream_FSM_done`	3	`compute_FSM_enable`
4	`ifmap_stream_FSM_done`	4	`ofmap_stream_FSM_enable`
5	`compute_FSM_done`	5	`ofmapRam_enable_acc`
6	`ofmap_stream_FSM_done`	6	`ifmapBuf_rst`

KernelStream FSM

The state diagram of the state machine which controls the input streaming of kernels is shown in figure 4.11, the inputs and outputs of the FSM are presented in table 4.4.

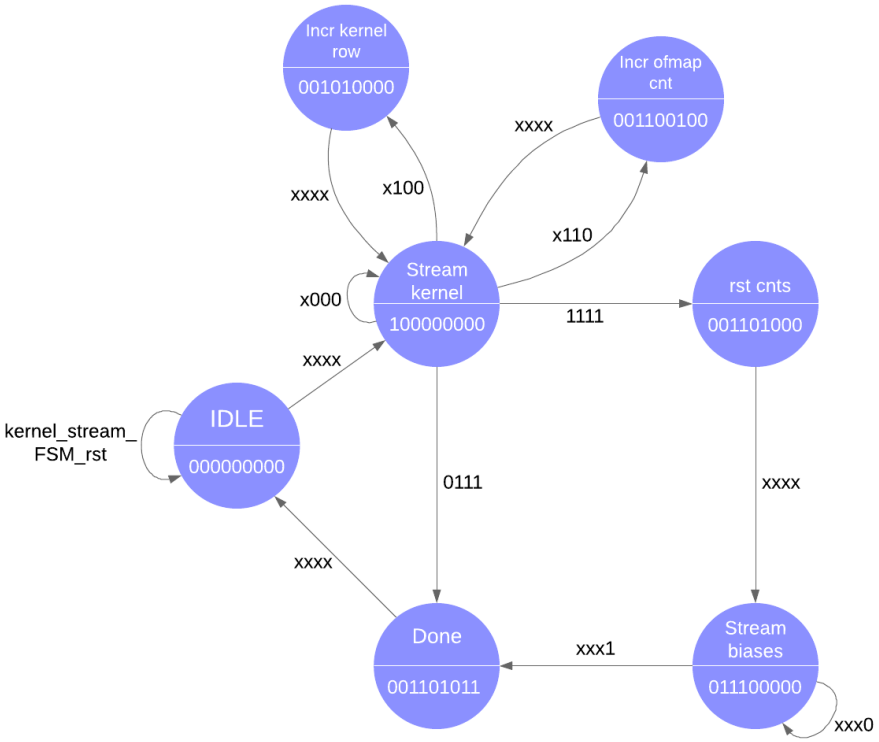


Figure 4.11: State diagram of the KernelStream FSM

Table 4.4: 2D KernelStream FSM inputs and outputs of FSM in figure 4.11

Inputs		Outputs	
#	Signal name	#	Signal name
1	stream_biases	1	kernel_stream_enable
2	counter_kernel_col_fin	2	bias_stream_enable
3	counter_kernel_row_fin	3	counter_kernel_col_rst
4	counter_no_ofmaps_fin	4	counter_kernel_row_rst
		5	counter_kernel_row_enable
		6	counter_no_ofmaps_rst
		7	counter_no_ofmaps_enable
		8	counter_ifmap_pixels_rst
		9	done

IfmapStream FSM

The state diagram of the state machine which controls the input streaming of ifmap pixels is shown in figure 4.12, the inputs and outputs of the FSM are presented in table 4.5.

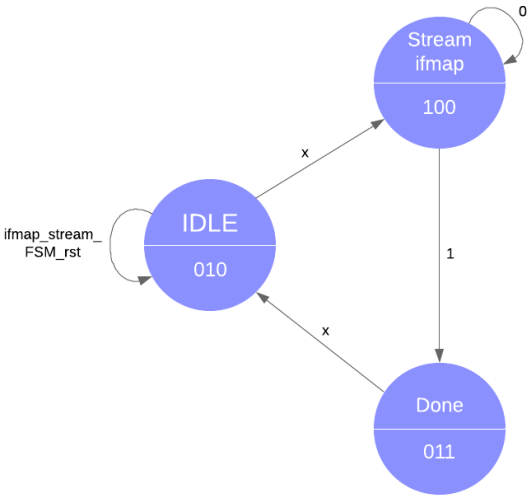


Figure 4.12: State diagram of the IfmapStream FSM

Table 4.5: 2D IfmapStream FSM inputs and outputs of FSM in figure 4.12

Inputs		Outputs	
#	**Signal name**	#	**Signal name**
1	`counter_ifmap_buffer_fin`	1	`ifmap_stream_enable`
		2	`counter_ifmap_buffer_rst`
		3	`done`

Computation FSM

The state diagram of the state machine which controls the computation is shown in figure 4.13, the inputs and outputs of the FSM are presented in table 4.6.

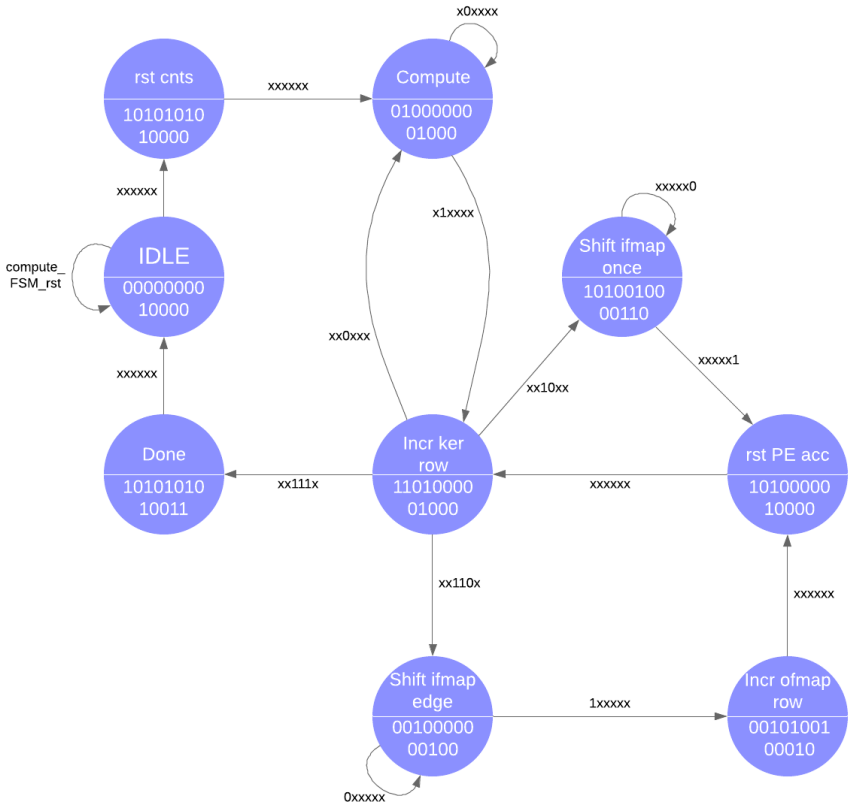


Figure 4.13: State diagram of the Computation FSM

Table 4.6: 2D Computation FSM inputs and outputs of FSM in figure 4.13

Inputs		Outputs	
#	**Signal name**	#	**Signal name**
1	`counter_kernel_col_fin`	1	`counter_kernel_col_rst`
2	`counter_kernel_col_fin_minus_1`	2	`counter_kernel_col_enable`
3	`counter_kernel_row_fin`	3	`counter_kernel_row_rst`
4	`counter_ofmap_width_fin`	4	`counter_kernel_row_enable`
5	`counter_ofmap_height_fin`	5	`counter_ofmap_width_rst`
6	`ifmapStream_handshake`	6	`counter_ofmap_width_enable`
		7	`counter_ofmap_height_rst`
		8	`counter_ofmap_height_enable`
		9	`PE_rst_acc`
		10	`PE_enable_acc`
		11	`ifmapStream_enable`
		12	`ofmapRam_write_enable`
		13	`done`

The states **Shift ifmap once** and **Shift ifmap edge** are both for shifting in new ifmap data while computation is going on. The **Shift ifmap once** state is for shifting the sliding

window (figure 2.3) one step along the same row. The **Shift ifmap edge** state is for switching rows, thus requiring to stream in `kernel_width` ifmap samples before proceeding.

OfmapStream FSM

The state diagram of the state machine which controls the output streaming of ofmaps is shown in figure 4.14, the inputs and outputs of the FSM are presented in table 4.7.

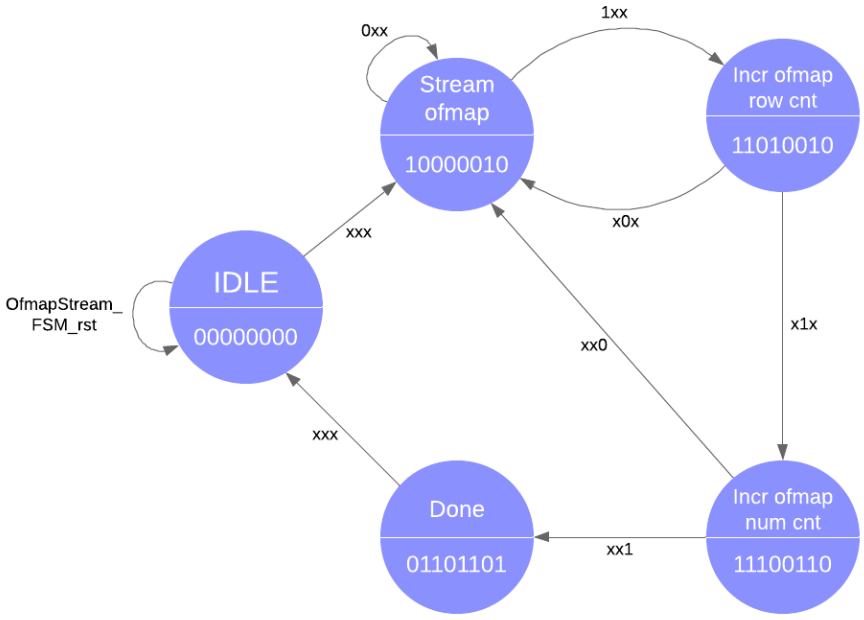


Figure 4.14: State diagram of the OfmapStream FSM

Table 4.7: 2D OfmapStream FSM inputs and outputs of FSM in figure 4.14

Inputs		Outputs	
#	**Signal name**	#	**Signal name**
1	`counter_ofmap_width_fin_minus_1`	1	`ofmap_stream_enable`
2	`counter_ofmap_height_fin`	2	`counter_ofmap_width_rst`
3	`counter_no_ofmaps_fin`	3	`counter_ofmap_height_rst`
		4	`counter_ofmap_height_enable`
		5	`counter_no_ofmaps_rst`
		6	`counter_no_ofmaps_enable`
		7	`ofmapRam_enable_acc`
		8	`done`

4.3 Verification

The 2D CNN accelerator has been more thoroughly verified than the 1D CNN accelerator. The basic method is the same (the one shown in figure 4.15), but one difference is that the fixed point software version of the 2D CNN accelerator is not truncating, but rounding. This is because the multiplication-accumulation in the PEs here are rounding instead of truncating, leading to a much smaller quantization error.

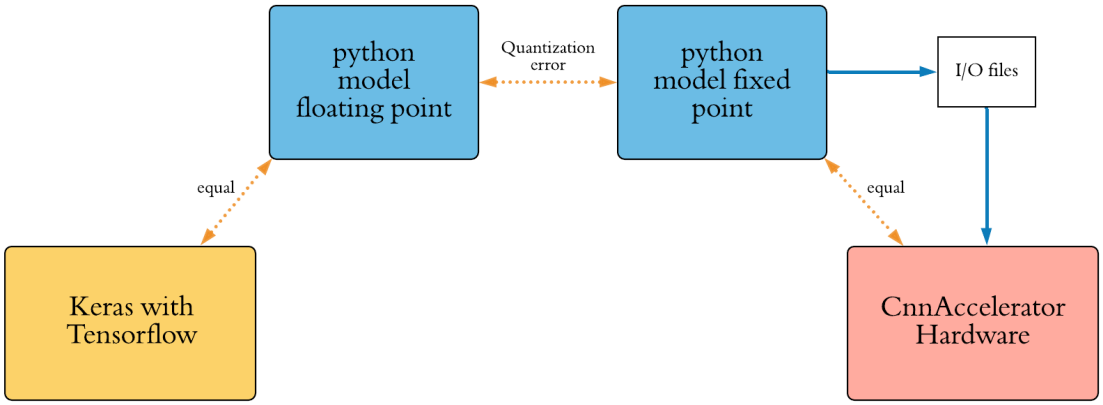


Figure 4.15: Functional verification method.

Testbench

The SystemVerilog testbench which is used to verify the functionality has the structure presented in figure 4.16.

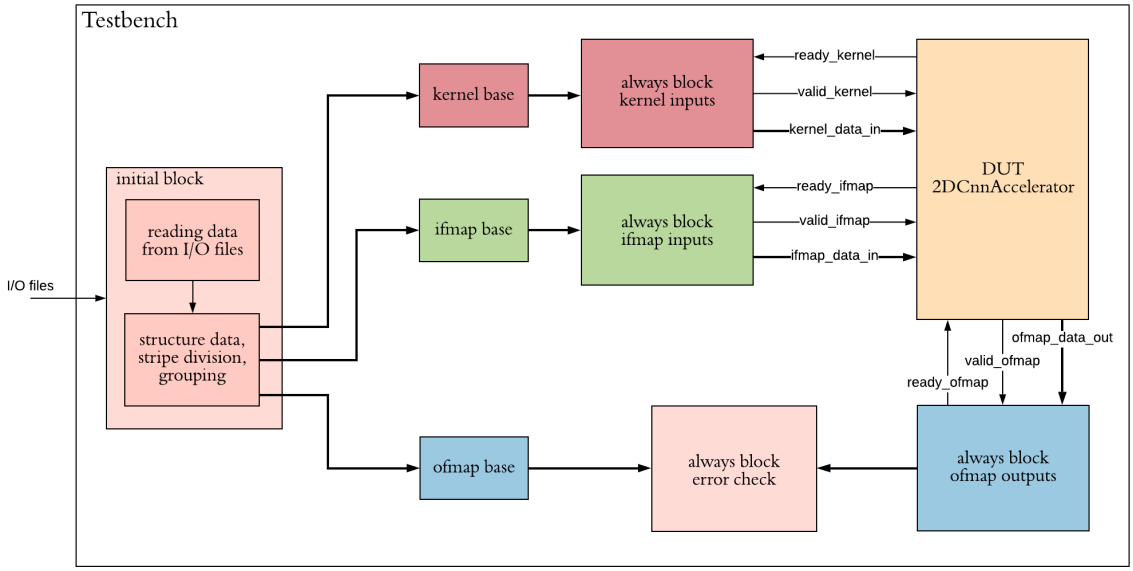


Figure 4.16: Structure of the testbench for the 2D CnnAccelerator. Biases have been omitted in figure as they are treated just like kernels

The initial block reads the files written by the SW implementation of figure 4.15. In the *ofmap base* the expected ofmap values are stored. These values are the ones written by the software fixed point version in figure 4.15. In the *error check* always block the ofmap values of the 2D CNN Accelerator are compared with the theoretically computed values of the ofmap base. Only when *all* values are equal will the test be considered a success.

The CNN accelerator is designed for being a low power, low area hardware accelerator. CNNs are often huge so it may be a problem to combine the two, for instance, the CNN discussed in section 4.3.3 has ifmap widths of 224 and hundreds of ofmaps as outputs in certain layers. In spite of being parametrized and configurable the CNN accelerator is going to have a limited maximal ifmap width and number of ofmaps. In sections 4.3.1 and 4.3.2 two important techniques to overcome these limitations are presented.

4.3.1 Stripe division

Take a case where the parameter B is smaller than the width of the ifmaps that are going to be processed. Say the ofmapMems have a size 8192 words leading to $B = 90$ (see section 4.2.3 for an explanation). Then say the ifmap width and height are 180 and that the kernel width is 3.

The stripe width w_s of ifmaps is given by:

$$w_s = \left\lfloor \frac{N_w}{h_i} \right\rfloor = \left\lfloor \frac{8192}{180} \right\rfloor = \lfloor 45.5 \rfloor = 45 \tag{4.4}$$

Where N_w is the number of words per RAM and h_i is the height of ifmaps. The important point is that the number of pixels in the stripe ($w_s \times h_i$) is less than or equal to N_w. The number of stripes n_s is given by:

$$n_s = \left\lceil \frac{w_i}{w_s - (w_k - 1)} \right\rceil = \left\lceil \frac{180}{45 - (3 - 1)} \right\rceil = \lceil 4.2 \rceil = 5 \tag{4.5}$$

Where w_i is the width of ifmaps, w_s is the stripe width and w_k is the kernel width.

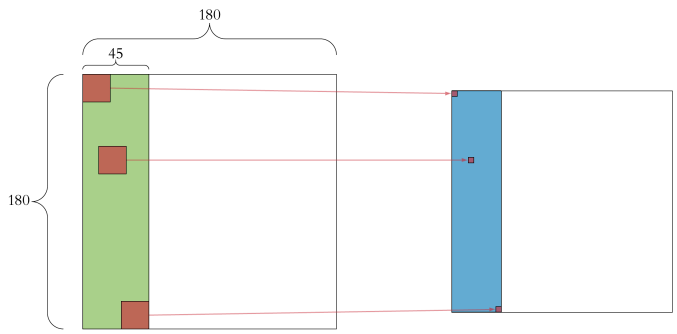


Figure 4.17: Ifmap stripe division with ifmap width of 180, RAM size of 8192 and $B = 90$.

All ifmaps are divided equally, all ifmaps send in the same stripe to the CNN Accelerator

before moving on to the next stripe shown in figure 4.18.

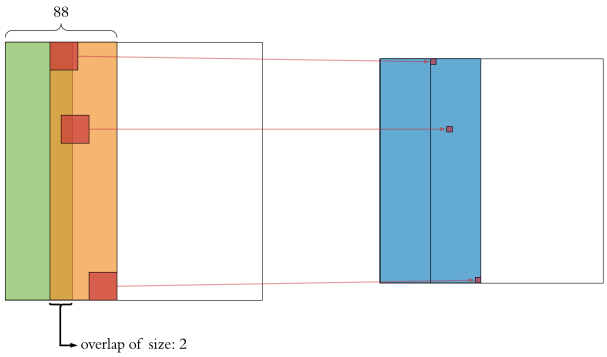


Figure 4.18: The second stripe, overlapping the first with 2 pixels.

The overlap of adjacent stripes is $w_k - 1$.

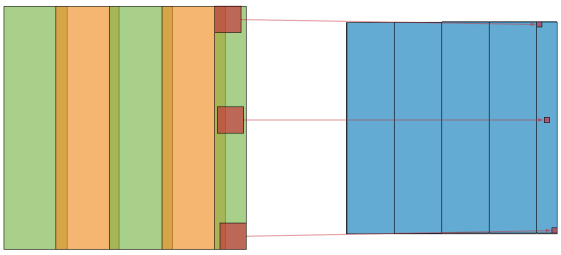


Figure 4.19: The last stripe

4.3.2 Ofmap grouping

Ofmap grouping is fairly simple compared to stripe division. It is done when the number of ofmaps that need to be processed is greater than M. Then one can quite simply just process M ofmaps at a time, and repeat successively until all have been processed. When one group of ofmaps has finished processing, the `layer_finished` signal needs to be set high as if one entire layer was finished. It can be seen as processing one layer as if it was several.

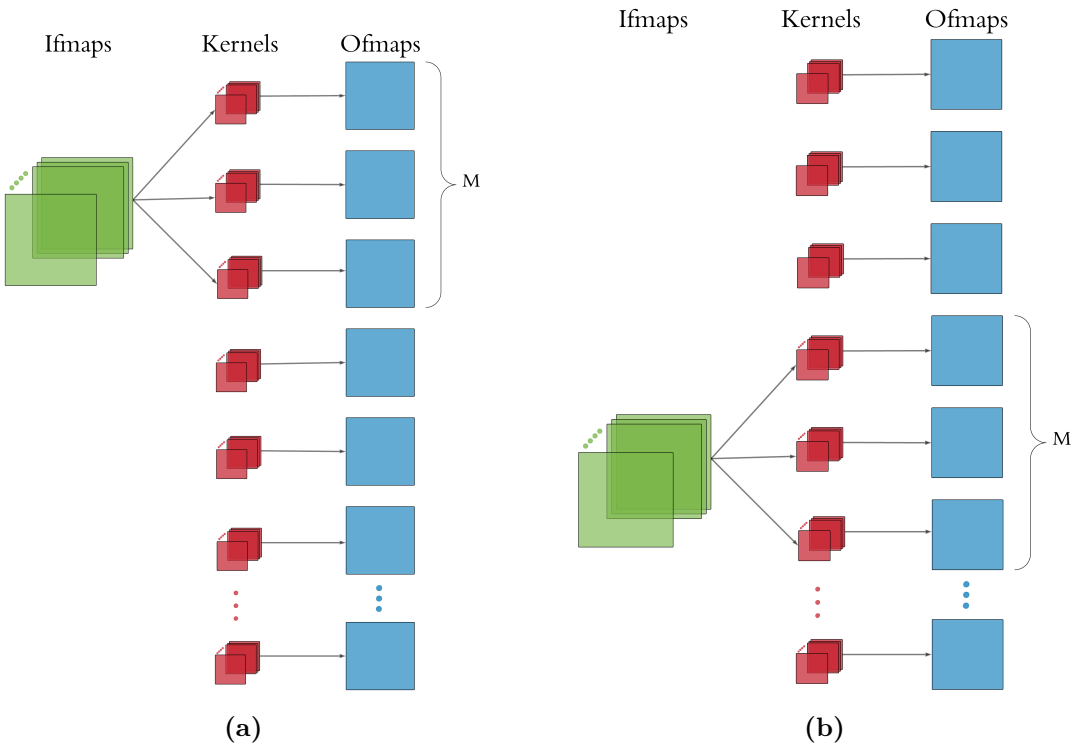


Figure 4.20: Example of ofmap grouping where $M = 3$ and the number of ofmaps is > 3

This technique will require sending in all ifmaps into the accelerator as many times as there are ofmap groups, resulting in a sub-optimal data reuse scheme. If ifmap stripe division and ofmap grouping both are necessary, then stripe division is performed within ofmap groups. So ofmap grouping is the highest level of hierarchy in the data preparation.

4.3.3 Benchmarking using VGG16

VGG16 [Kar14] is a famous CNN that has been among the best in image recognition for years. It has been chosen for benchmarking in this project because a Python model using Keras and Tensorflow is readily available and in [YuH17b] (Eyeriss) it has been used for benchmarking (see table VI in [YuH17b]). The method used is the following: firstly an image of the ImageNet database is chosen at random, then that image is used as the input of the VGG16 network. Inference is run so that the image is classified and the CNNs functionality is confirmed. Then one layer is chosen and the ofmaps of that layer are fetched and written to files. This constitutes the yellow box of figure 4.15. Then the rest of the process proceeds as described in section 3.3.

Table 4.8: The conv layers of VGG16 that have been used for verification and benchmarking. All ifmap widths are 2 greater than whats normally given in VGG16, this is because all ifmaps in VGG16 are zero padded outside the borders with a width of 1

Layer	**ifmaps**	**ofmaps**	**ifmap width**	**kernel width**
CONV1-1	3	64	226	3
CONV1-2	64	64	226	3
CONV2-1	64	128	114	3
CONV2-2	128	128	114	3

The accelerator has been benchmarked using two different parameter configurations one having ($B = 90$, $K = 5$, $M = 5$) and ($B = 90$, $K = 5$, $M = 32$). The testbench has been implemented using both techniques described in sections 4.3.1 and 4.3.2 as the dimensions of the VGG16 CNN are quite a bit larger than these two given parameter configurations.

4.4 Results

VGG16

The first 4 conv layers of VGG16 (a CNN presented in section 4.3.3) have been implemented successfully on the 2D CNN accelerator. The results of processing these layers with two different parameter configurations are shown in tables 4.9 and 4.10. The tests have been run using the method described in section 4.3.

Table 4.9: Processing of VGG16. ($M = 5$, $B = 90$, $K = 5$ and 8k RAMs), at 100MHz and 1V supply voltage. Area is 585 KGEs. Power is estimated using Spyglass Power.

Layer	Power (mW)	Time (ms)	Time (ck cycles)
CONV1-1	2.3	249.8	25.0 mill
CONV1-2	2.3	4675	468 mill
CONV2-1	2.4	2333	233 mill
CONV2-2	2.5	4650	465 mill

Table 4.10: Processing of VGG16. ($M = 32$, $B = 90$, $K = 5$ and 8k RAMs) at 100MHz and 1V supply voltage. Area is 3.4 MGEs. Power is estimated using Spyglass Power.

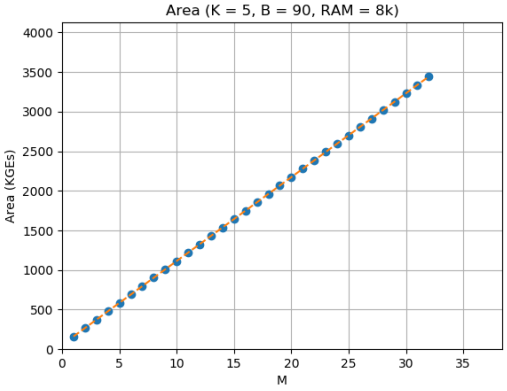
Layer	Power (mW)	Time (ms)	Time (ck cycles)
CONV1-1	26.7	65.7	6.57 mill
CONV1-2	26.5	749	74.9 mill
CONV2-1	28.2	374	37.4 mill
CONV2-2	28.7	732	73.2 mill

Power

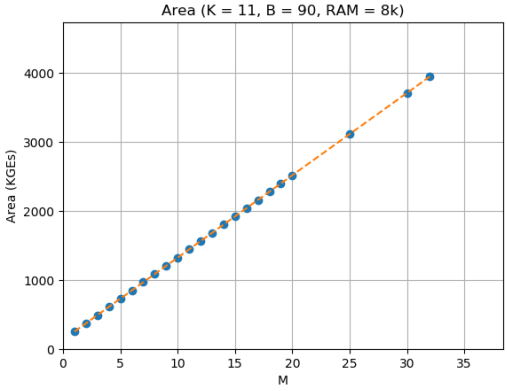
Power has been estimated using Synopsys Spyglass power (chapter 1). The results of Spyglass Power are scenario based, meaning that the power consumption estimate is based on the activity and switching in a specific testbench. The results of Spyglass Power in some specific VGG16 layers are shown in the tables 4.9 and 4.10.

Area

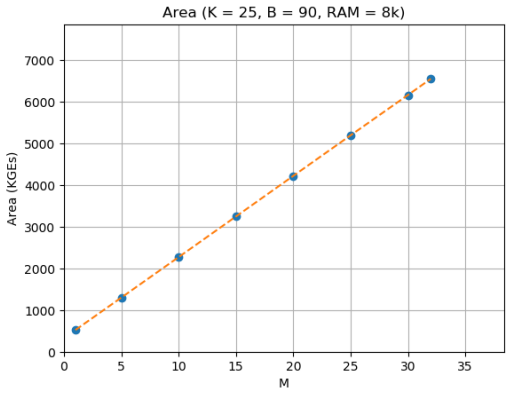
The accelerator has been synthesized with many different parameter configurations, some plots of area as a function of different parameters are shown on the following pages. The blue dots represent results of syntheses that have been carried out and the orange lines are the interpolations between them. Area is measured in *Kilo/Mega Gate Equivalents* (KGEs, MGEs respectively).



(a) $K = 5$, $B = 90$, $ram\_size = 8k$.

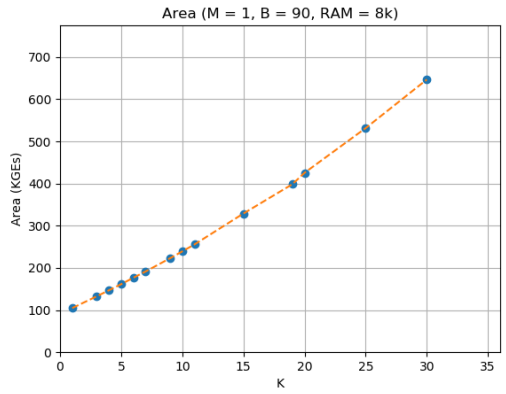


(b) $K = 11$, $B = 90$, $ram\_size = 8k$.

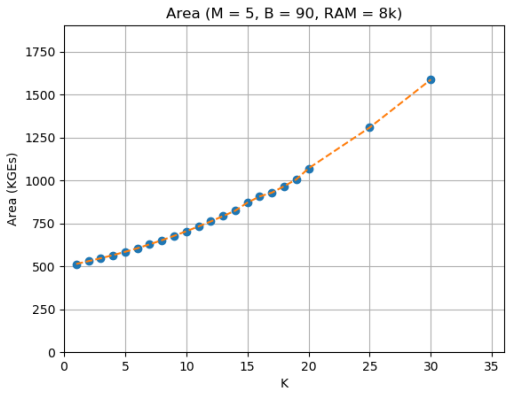


(c) $K = 25$, $B = 90$, $ram\_size = 8k$.

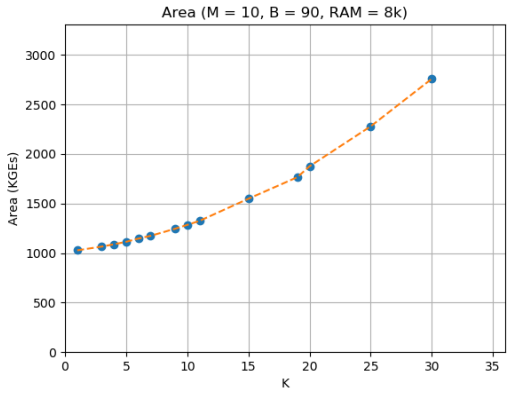
Figure 4.21: Area plotted as a function of M. The number of OfmapMems and the number of PEs



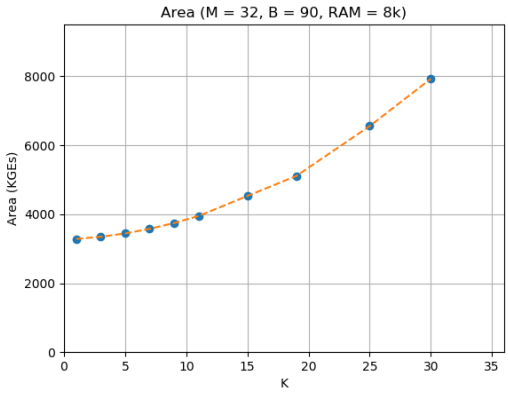
(a) $M = 1$, $B = 90$, $ram\_size = 8k$.



(b) $M = 5$, $B = 90$, $ram\_size = 8k$.

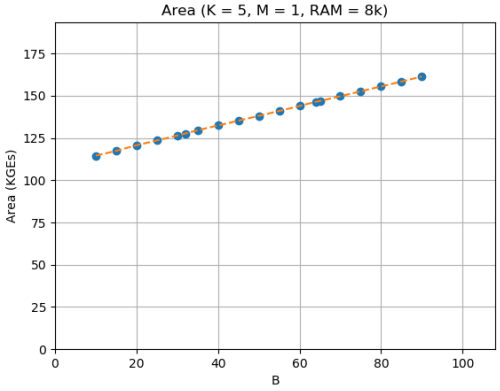


(c) $M = 10$, $B = 90$, $ram\_size = 8k$.

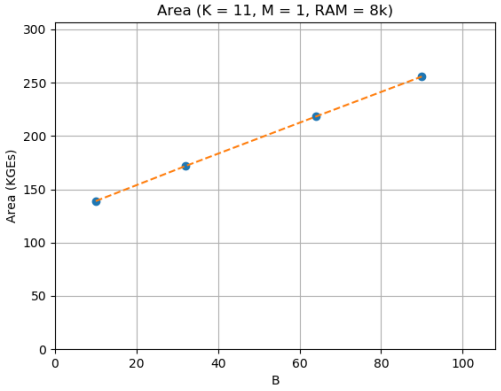


(d) $M = 32$, $B = 90$, $ram\_size = 8k$.

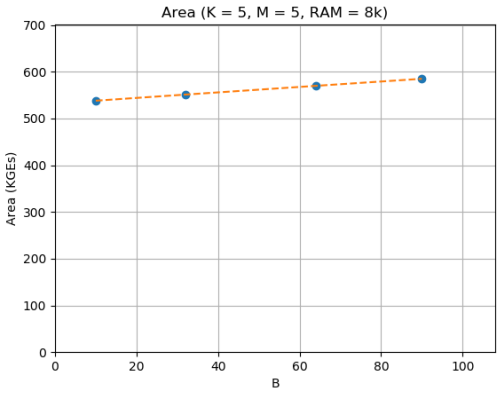
Figure 4.22: Area plotted as a function of K. The maximal kernel size.



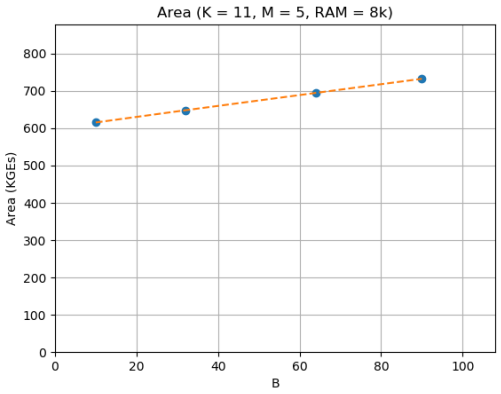
(a) $K = 5$, $M = 1$, $ram\_size = 8k$.



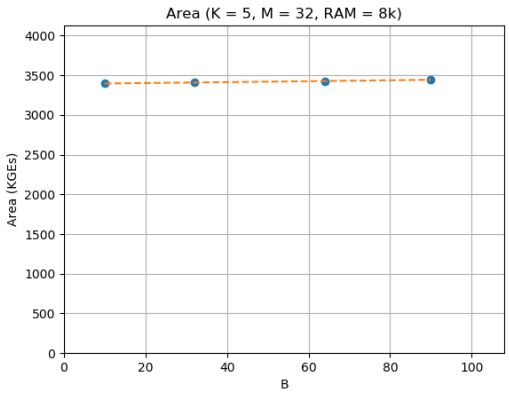
(b) $K = 11$, $M = 1$, $ram\_size = 8k$.



(c) $K = 5$, $M = 5$, $ram\_size = 8k$.



(d) $K = 11$, $M = 5$, $ram\_size = 8k$.



(e) $K = 5$, $M = 32$, $ram\_size = 8k$.

Figure 4.23: Area plotted as a function of B. The maximal ifmap width

Section A.3 shows some 3D plots of area as a function of 2 parameters simultaneously.

4.5 Discussion

4.5.1 Area

The plots of area presented in the results, and section A.3 in the appendix, show how the area varies as a function of the parameters M, K and B. It is quite clear that M is dominating the area. This means that the number of PEs and OfmapMems have the greatest impact on area. This is shown by the high intercepts of the vertical axis in figures 4.22 and 4.23 when M gets large.

The parameter K, when large enough, will also affect the area significantly. The plots have a parabolic shape, as the kernel registers in the PEs are squares. And it will also affect the number of rows in the IfmapBuffer, making the increase even steeper. When K is larger, B has a greater impact on the area this is shown in figure 4.23(b) when compared to figure 4.23(a).

4.5.2 Estimation of speed

Simulation of the 2D CNN accelerator can take hours when processing a large conv layer. Therefore this section aims to produce calculations that estimate how much time the accelerator uses to process a given conv layer. Using pure calculation instead of simulation. Also knowing how much time the CNN accelerator uses to compute is related to computing the number of operations or MACs (Multiply-Accumulates) performed (shown in section 4.5.3).

Computation time

If `kernel_width` $\leq K$, `no_ofmaps` $\leq M$ and `ifmap_width` $\leq B$ the time used for computation can quite simply calculated from formula (4.6).

$$T_{comp\_simple} = ((w_k^2 + 2) \cdot w_o \cdot h_o) \cdot n_i \tag{4.6}$$

Where T is total time taken, measured in number of clock cycles, w_k is the kernel width, w_o is the ofmap width, h_o is the ofmap height and n_i is the number of ifmaps. This applies only of $n_o \leq M$ and $w_k \leq K$ and $h_o \cdot w_o \leq mem\_size$. The +2 inside the parentheses comes from the two extra clock cycles spent by the PE when waiting for new ifmap values (described in section 3.2.7).

Computation time including stripe division

If however $w_i > B$ stripe division must be applied and the following formulae are obtained.

$$sw_i = min\Big(min(\left\lfloor \frac{mem\_size}{h_i} \right\rfloor, w_i), B\Big) \tag{4.7}$$

$$sw_o = sw_i - (w_k - 1) \tag{4.8}$$

Where sw_i is the stripe width of ifmap stripes, sw_o is the stripe width of the ofmap stripes and $min(a, b)$ is the function returning a if $a < b$ and b otherwise. The number of stripes per feature map thus becomes:

$$n_s = \left\lceil \frac{w_o}{sw_o} \right\rceil \tag{4.9}$$

$$T_{comp} = (((w_k^2 + 2) \cdot sw_o \cdot h_o) \cdot n_i) \cdot (n_s - 1) + (((w_k^2 + 2) \cdot last\_sw_o \cdot h_o) \cdot n_i) \tag{4.10}$$

Which simplifies to:

$$T_{comp} = (((w_k^2 + 2) \cdot h_o) \cdot n_i) \cdot ((n_s - 1) \cdot sw_o + last\_sw_o) \tag{4.11}$$

Computation time including ofmap grouping

If $n_o > M$ ofmap grouping must be applied and the following formulae are obtained.

$$n_g = \left\lceil \frac{n_o}{M} \right\rceil \tag{4.12}$$

$$T_{comp} = (((w_k^2 + 2) \cdot h_o) \cdot n_i) \cdot ((n_s - 1) \cdot sw_o + last\_sw_o) \cdot n_g \tag{4.13}$$

Where $last\_sw_o = w_o \pmod{sw_o}$ and denotes the width of the last stripe, as it will be less than or equal to sw_o the other stripes. Here the situation where w_o is divisible by sw_o is ignored because the situation does not occur in the given scenarios and it would complicate these calculations further. Time taken during edge shifts (when the sliding window goes to the next row) is also omitted. This is the final version of the computation time that will be used in calculations.

Streaming time

The time for streaming inputs (both weights and ifmaps) is given in the following formulae. t is the input streaming time for a single cycle of the accelerator's state machine. Biases

have been omitted as they take a negligible amount of time to stream.

$$t = K \cdot sw_i + w_k^2 \cdot M \tag{4.14}$$

$$T_{stream\_in} = n_i \cdot n_s \cdot n_g \cdot t \tag{4.15}$$

Time taken streaming ofmaps out is given in the following formula.

$$T_{stream\_out} = w_o \cdot h_o \cdot n_o \tag{4.16}$$

Total time

The total time used for processing a conv layer is given in the following formula.

$$T_{tot} = T_{comp} + T_{stream\_in} + T_{stream\_out} \tag{4.17}$$

Where T_{comp} is given in formula (4.13), T_{stream_in} in formula (4.15) an and T_{stream_out} in (4.16).

Using the above calculations an estimate of the speed of the 2D CNN accelerator can be made, given the hardware parameters and the specific conv layer being implemented.

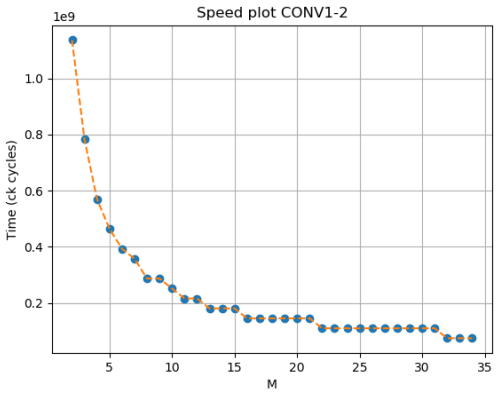
Table 4.11: conf1: $(B = 90, K = 5, M = 5)$, **conf2:** $(B = 90, K = 5, M = 32)$, both use OfmapRAMs of size 8k. CONV1-2 and CONV2-1 refer to two of the conv layers of VGG16, described in section 4.3.3.

	conf1		**conf2**	
	CONV1-2	**CONV2-1**	**CONV1-2**	**CONV2-1**
w_k	3	3	3	3
w_i	226	114	226	114
h_i	226	114	226	114
w_o	224	112	224	112
h_o	224	112	224	112
n_i	64	64	64	64
n_o	64	128	64	128
n_s	7	2	7	2
sw_i	36	71	36	71
sw_o	34	69	34	69
$last\_sw_o$	20	43	20	43
n_g	13	26	2	4
Speed (cycles)	**464 mill**	**232 mill**	**74 mill**	**37 mill**

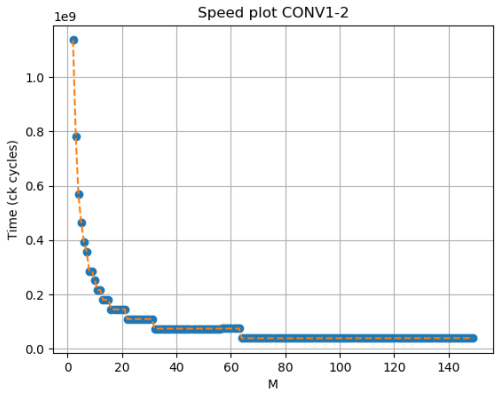
The last row in table 4.11 indicates the calculated estimate of speed using the above calcu-

lations given the configurations and implemented conv layer. If compared with the results from simulations in tables 4.9 and 4.10 it can be seen that the numbers are quite accurate, but slightly lower, that is because of the simplifications done in these estimates.

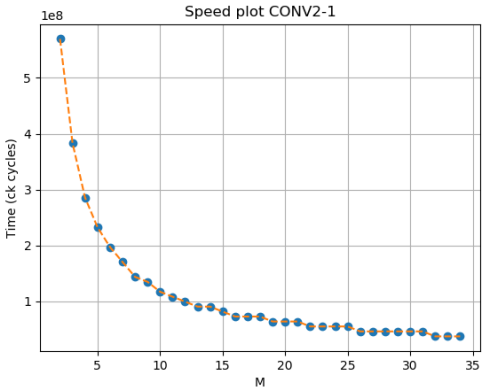
The following plots show how the speed (time taken) varies with the hardware parameter M which is equal to the number of PEs and OfmapMems. The plots are based on the calculations made in this section. The higher M is, the faster it should go, as reflected in the results.



(a) Implementing CONV1-2 of VGG16.



(b) CONV1-2 of VGG16. Greater range M



(c) CONV2-1 of VGG16

Figure 4.24: Plots of the time used processing layers of VGG16 as a function of M. $K = 5$, $B = 90$ and 8k RAMs for all plots.

As can be seen in the plots, the speed makes discrete jumps at certain places. This happens when the value of M has increased enough that n_g decreases, meaning that fewer ofmap groups are required and parallelization can increase. At $M = 64$ it can be seen that the speed stagnates, as no further increase in M can give fewer ofmap groups.

4.5.3 Estimation of Energy efficiency

Estimation of number of operations (MACs)

The number of operations that need to be performed in a conv layer will depend on the implementation. This section will discuss how the presented architecture performs with the two different parameter configurations used in the results section (section 4.4). The metric that will be used to measure number of operations is MACs (multiply and accumulate operations). Assume that all M PEs are active when processsing, then the number of operations that take place at every ifmap position (window position, see figure 2.3) is the following.

$$N_{MACs\_1} = M \cdot w_k^2 \tag{4.18}$$

That happens for all ifmap positions in the given stripe resulting in the following number of MACs per stripe.

$$N_{MACs\_2} = M \cdot w_k^2 \cdot h_o \cdot sw_o \tag{4.19}$$

That happens for all stripes in all ifmaps resulting in the following formula, but the last ifmap stripe has a smaller width.

$$N_{MACs\_3} = M \cdot w_k^2 \cdot h_o \cdot n_i \cdot (sw_o \cdot (n_s - 1) + last\_sw_o) \tag{4.20}$$

Then that happens for all ofmap groups resulting in formula (4.21) which is an estimate of the total number of MACs that take place in a complete conv layer.

$$N_{MACs} = M \cdot w_k^2 \cdot h_o \cdot n_i \cdot (sw_o \cdot (n_s - 1) + last\_sw_o) \cdot n_g \tag{4.21}$$

Where M is the hardware parameter presented, w_k is the kernel width, sw_o is the stripe width, $last\_sw_o$ is the width of the last ifmap stripe, h_o is the ofmap height, n_s is the number of stripes, n_i is the number of ifmaps and n_g is the number of ofmap groups. The additions of `offset_in` in the PEs has been ignored, only the multiply-accumulate operations are considered. The addition of biases in the output of the 2D CNN accelerator have also been ignored.

Energy Efficiency (MACS/s/W)

The power used for the two different parameter configurations in the four different conv layers is shown in the results 4.4 in tables 4.9 and 4.10. The calculation of the number of MACs per time per power (MACs per unit energy) is shown in the following formula.

$$EnEff = \frac{\frac{\#MACs}{t}}{P} \tag{4.22}$$

Where $EnEff$ is the energy efficiency, $\#MACs$ is the number of MACs performed, t is the time taken and P is the average power consumed.

Table 4.12: conf1: $(B = 90, K = 5, M = 5)$, **conf2:** $(B = 90, K = 5, M = 32)$, both use OfmapRAMs of size 8k. The ck frequency is 100MHz and 1V supply voltage

Layers	Configuration	MACs	Time (ms)	Power (mW)	Energy Efficiency (MACs/s/W)
CONV1-1	**conf1**	88.1 mill	249.8	2.3	153 bill
	conf2	86.7 mill	65.7	26.7	49.4 bill
CONV1-2	**conf1**	1.88 bill	4675	2.3	175 bill
	conf2	1.85 bill	749	26.5	93.2 bill
CONV2-1	**conf1**	939 mill	2333	2.4	168 bill
	conf2	925 mill	374	28.2	87.8 bill
CONV2-2	**conf1**	1.88 bill	4650	2.5	162 bill
	conf2	1.85 bill	732	28.7	88.1 bill

According to the calculations shown in table 4.12, the different parameter configurations does not have a great impact on the number of MACs performed. In fact, they should not have much effect at all on the number of operations performed. The subtle differences come from inaccuracies in the given calculations. The punishment of having low parallelization does not come in the form of having to do more operations, but having to do them over a longer period of time.

4.5.4 Comparison with Eyeriss

Eyeriss [YuH17b] is a state-of-the-art CNN hardware accelerator that can be considered comparable to the presented architecture. It processes feature maps and weights in 16-bits fixed point and has been benchmarked using the CNN VGG16 [Kar14].

Table 4.13: Comparison between Eyeriss and the presented architecture implementing layers of VGG16. **conf1** refers to the presented architecture with parameters ($K = 5$, $M = 5$, $B = 90$, $ram\_size = 8k$) and **conf2** refers to ($K = 5$, $M = 32$, $B = 90$, $ram\_size = 8k$). All architectures use 1V supply voltage. Eyeriss runs at 200MHz and the presented accelerator at 100MHz.

Layer	Architecture	Power (mW)	Time (ck cycles)	Area (MGEs)	EnEff (MACs/s/W)
CONV1-1	**conf1**	2.3	25.0 mill	0.6	153 bill
	conf2	26.7	6.57 mill	3.4	49.4 bill
	Eyeriss	247	7.6 mill	8.0	27.7 bill
CONV1-2	**conf1**	2.3	468 mill	0.6	175 bill
	conf2	26.5	74.9 mill	3.4	93.2 bill
	Eyeriss	218	182 mill	8.0	31.4 bill
CONV2-1	**conf1**	2.4	233 mill	0.6	168 bill
	conf2	28.2	37.4 mill	3.4	87.8 bill
	Eyeriss	242	94.1 mill	8.0	28,2 bill
CONV2-2	**conf1**	2.5	465 mill	0.6	162 bill
	conf2	28.7	73.2 mill	3.4	88.1 bill
	Eyeriss	231	178.9 mill	8.0	29.6 bill

The time and power values of Eyeriss are taken from table VI in [YuH17b]. Its energy efficiency is not directly given in that table so it has been calculated using formula (4.22) based on the values given in the table. The performance of both configurations of the presented architecture is significantly better than Eyeriss in terms of energy efficiency in the given conv layers of VGG16. The maximal energy efficiency of Eyeriss is stated to be 122.8 GMACs/s/W at 0.82 V. Optimizing the supply voltage could also make the presented architecture more energy efficient. One final consideration is that Eyeriss is implemented in 65 nm CMOS and the presented design on 55 nm. This has implications regarding power consumption.

Discrepancy in number of MACs

In table VI in [YuH17b] the number of MACs per conv layer are shown. There is a discrepancy between the number of MACs performed in the conv layers of VGG16 by Eyeriss and by the presented accelerator. The presented accelerator consistently uses about a factor 3 fewer MACs than Eyeriss does. It is unknown why this discrepancy is there, it could be that Eyeriss computes the conv layers less efficiently using more operations.

Chapter 5

Discussion

5.1 Matrix multiplication

The presented architecture is essentially a collection of multiply-accumulate processors (PEs), each with memory (OfmapMems) and a distributor of data (IfmapBuffer). This architecture could quite easily be adapted to perform many other kinds of operations like dot products and matrix multiplications. To perform a matrix multiplication like: $\mathbf{M}_1 \times \mathbf{M}_2 = \mathbf{M}_3$, the columns of $\mathbf{M}_2$ can be loaded into the PE as though they were kernels, and the rows of $\mathbf{M}_1$ could be sent through the IfmapBuffer. When processing this could result in a sum of all element-wise products of rows and columns in $\mathbf{M}_1$ and $\mathbf{M}_2$ respectively, resulting in a matrix product. This could be done in both the 1D and 2D accelerators. The parameters M and K would limit the sizes of the tolerated matrices.

The main operation performed in inference in fully connected neural networks is a product of a matrix and a vector. Therefore this could conceivably be a starting point on how to accelerate fully connected NNs.

5.2 Quantization, range and precision

The verification method presented in sections 3.3 and 4.3 involve using a version of the CNN accelerator implemented in software using floating point numbers (leftmost blue box in figure 4.15). When the Keras model and this sofware version of the accelerator are equal, this is regarded as definitive proof that the accelerator computes a conv layer correctly. However there is always some error between the Keras model and the software implementation of the accelerator. The error was found to decrease when few ofmaps were accumulated (around $\sim 10^{-10}$ absolute value) and increase when more were accumulated (around $\sim 10^{-3}$ absolute value). So the error is believed to be only a result of small imperfections in the floating point number representation.

The difference between the floating point representation and fixed point representation give the quantization error and that is the largest error. In the truncating multiplications used in the 1D CNN accelerator the error quickly became disastrous, in some cases 100% off if the kernel is large and many ofmaps are accumulated. For the rounding operations however the error became significantly smaller, the maximal observed errors were <1% off even when many ofmaps are accumulated.

The quantization of both inputs and weights is a major concern in this work. The inputs and weights have been quantized in fixed point using 16 bits throughout this thesis. However the data widths are parameterized so that more or less precision and range can be supported depending on what may be needed. There is an often applied regularization technique known as *weight decay* (see chapter 3 in [Mic15]) that during training makes the NN prefer smaller weights over lower weights. This leads to the weights rarely being greater than one, and makes precision more important than range, so in VGG16 the weights were quantized with 16 bits having 14 fractional bits [16, 14] in two's complement. Data however is often pixels, and pixels are typically bytes, thus an integer of 8 bits between 0 and 255. As the processing gets deeper into the neural network then there is no upper limit at 255 anymore, therefore more bits are required, and also some precision. In VGG16 16 bits were used for data whereas 2 bits were fractional [16, 2] in two's complement.

Binary Weights

Some theoretical groundwork should be laid on how many data and weight bits are required, this could then be implemented and used to improve the accelerator's performance significantly, perhaps going as far as binary weights. Binary weights is a heavily explored field [M C16b] and often means allowing all weights to be only either -1 or 1. This gives huge hardware benefits, in terms of speed, area and power as all multipliers can be replaced by multiplexers.

This has not been explored thoroughly in this thesis because of the desire for versatility. This thesis does not limit itself to only one type of CNN. However the number of bits in the weights can be reduced by adjusting the parameter W_w, and the multipliers in the PEs can be replaced by more efficient operations. Binary weights make the idea of the Parallel PE (section 3.5.1) more attractive as the parallel multiplications would become less power consuming and area occupying. Because of this, the presented architecture is quite far behind YodaNN [Ren16] and Hyperdrive [Ren18] which both exceed 1 TOPs/s/W (although Ops are not the same as MACs, bear in mind that no multiplication is needed in a binary CNN, rendering MACs unsuitable for comparison). YodaNN and Hyperdrive however can only implement certain specialized kinds of CNNs with binary weights.

5.3 Control path

First and foremost this thesis presents an *architecture*. The control path created along with this architecture could be adjusted to handle some things more elegantly. For instance the `new_layer/layer_finished` logic could be replaced by having for example `no_ifmaps` as an input and thus not rely on the external user setting those two signals correctly at the right time. This would also eliminate the need for the latch in the presented FSMs (section 3.2.7). Given the logic that has been used, the latch could also have been eliminated using two outputs from MasterControlFSM and use them as *write_enable* and data input to a flip flop that resides outside of the FSM. The flip flop could then contain the `ofmapMem_enable_acc` signal.

Different stride lengths can also be supported. This can be done by having `stride_length` as a top level input and use a counter that counts to this value when streaming ifmap values while computing, when the ComputationFSM is active. ResNet [Kai15] for example, the best CNN around per 2018, uses larger stride lengths as a replacement for pooling.

State encoding in FSMs has not been considered in this thesis because again, the architecture is in focus here, not the control path. It affects power consumption, so it should be accounted for.

If padding around the ifmap borders should be applied (as VGG16 does), then the data is assumed to be padded outside the accelerator, support can be added for built-in padding, but this has not been considered.

5.4 Optimizations and variations

There are many optimizations that can be conceived. In the paragraphs that follow, some of them are discussed.

For example the `offset_in` adder in the PEs, see figure 3.5, could be eliminated if the possibility of loading a value directly into the *Acc reg* was introduced. Then the initialization phase of the PEs could consist of loading the `offset_in` value into the register. A method similar to this could also be used to add the biases directly into the PEs instead of needing the extra bias adder at the end of the top level architecture, see figure 3.3.

Latch based implementations of memory have been used in [Ren16] and [Ren18] to be able to scale voltage down further than a RAM or a register would allow. This could be investigated if voltage scaling is going to be done.

In the presented design only the ReLU activation function (formula 2.6) has been considered. To improve versatility, other activation functions could also be implemented. It could possibly be done generically by having a re-writable look up table as the activation function.

An algorithm like simulated annealing could be carried out for optimizing the hardware parameters for all relevant metrics.

Some sparsity optimizations can be done. So that when for example a 0 arrives to the PEs from the IfmapBuffer, the PEs do nothing, save some energy, and skip to the next ifmap sample/pixel.

The OfmapMem sizes (section 4.2.3) could be increased to relax its constraint on B somewhat, then the area and power must thus be evaluated for the greater OfmapMems. The IfmapBuffer could then also grow larger, and it could be an advantage to implement it also as a RAM.

Considering the 1D CNN accelerator. The operations involved in that could also be done by the 2D CNN accelerator. All the necessary resouces are present, only some hardware would remain unused. In order to do this some modifications to the 2D CNN accelerator's control path must be done.

Layout and physical production have not been done, so many considerations that are involved when implementing hardware on chip have not yet been taken. The estimates given by Synopsys Design compiler and Synopsys Spyglass might therefore not necessarily represent how the hardware would behave on chip.

5.5 Future work

One thing that has not been elaborated on as much as planned is the power consumption. Time has not allowed further investigation into what dominates the power consumption and thus neither the energy efficiency. A future goal could be to find the optimal point in the speed vs power trade-off, the point where the energy efficiency is maximal. Thus far only the results of a few cases have been stated, it has not been evaluated in its entirety.

This thesis is only concerned with accelerating the convolutions involved in CNN. However fully connected layers also play an essential role in CNNs and NNs in general. It would be very powerful if an accelerator for fully connected NNs were to be designed and together they could perform all operations necessary in a CNN. There are many problems that arise with respect to accelerating fully connected NNs, like the sheer number of weights involved (much higher than in a conv layer), therefore much work has been done on pruning weights and utilizing sparsity in NNs (EIE for example [Son16]). However, [YuH18b] says the following about sparsity: "While this is efficient for sparse DNNs, there would be significant overhead for processing dense DNNs in the compressed format. This is a challenge, since there is no guarantee of sparsity in the DNN.".

Chapter 6

Conclusion

This thesis has presented a configurable, versatile and flexible architecture for hardware acceleration of *convolutional neural networks* (CNNs). The goal was to make a hardware accelerator that is able to process any CNN. This has been achieved by making an accelerator that processes one convolutional layer at a time so no constraints are put on the depth of the network. The accelerator has been made versatile so that any kernel sizes, feature map sizes and number of feature maps both in and out can be processed.

The thesis has presented an accelerator for a less common type of CNN, namely one-dimensional CNNs. That architecture was subsequently further developed to design another accelerator that processes two-dimensional CNNs. The two-dimensional CNN accelerator has been tested and compared with literature and found to be comparable to state-of-the art CNN hardware accelerators. This was found by comparison with Eyeriss [YuH17b] (a state-of-the-art CNN accelerator in terms of energy efficiency) in implementing the CNN VGG16 [Kar14] in 16 bit fixed point (for both weights and feature maps). The presented architecture has been implemented mainly with two different sizes, one large (occupying 3.4 MGEs (*Million Gate Equivalents*) of area) with high levels of parallelism and one smaller (0.59 MGEs) with low levels of parallelism. The former processes VGG16 consuming an average power of about 27 mW achieving between 49 and 93 GMACs/s/W as energy efficiency. The latter processes VGG16 slower, but consumes only around 2.4 mW of power and has an energy efficiency between 153 GMACs/s/W and 175 GMACs/s/W. Both with a 100MHz clock and 1V supply voltage. When compared to Eyeriss' maximal efficiency of 122.8 GMACs/s/W at 0.82V while occupying 8 MGEs, the presented architecture is a significant improvement. For some select convolutional layers, the energy efficiency is improved by a factor of more than 5.

Bibliography

[Abh18] Abhinav Ralhan. *Self Organizing maps*. 2018. URL: https://towardsdatascience.com/self-organizing-maps-ff5853a118d4 (visited on 2018-12-13).

[Ale12] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: (2012).

[Cat17] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose and James S. Plank. "A Survey of Neuromorphic Computing and Neural Networks in Hardware". In: *arXiv: 1705.06963* (2017).

[Chr18] S. T. Christensen. *A literature review of low power hardware accelerators for neural networks*. Tech. rep. Norwegian University of Science and Technology, 2018-12.

[Dav18] M. Davies. "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning". In: *IEEE micro* 38.1 (2018), pp. 82–99.

[Dem17] Demis Hassabis, David Silver. *AlphaGo Learning from scratch*. 2017. URL: https://deepmind.com/blog/alphago-zero-learning-scratch/ (visited on 2018-12-13).

[Kai15] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. "Deep Residual Learning for Image Recognition". In: *arXiv 1512.03385* (2015).

[Kar14] Karen Simonyan, Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *Proceedings of the IEEE* (2014), pp. 1–14.

[Ker19] Keras. *Keras*. 2019. URL: https://keras.io/ (visited on 2019-06-15).

[Li 18] Li Fei-Fei. *ImageNet*. 2018. URL: http://www.image-net.org (visited on 2019-05-20).

[Luc19] Lucidchart. *Lucidchart*. 2019. URL: https://www.lucidchart.com (visited on 2019-06-08).

[Luk15] Lukas Cavigelli, M. Magno, and Luca Benini. "Accelerating real-time embedded scene labeling with convolutional networks". In: *in Proceedings of the 52nd Annual Design Automation Conference* (2015).

[M C16a] M. Courbariaux and Y. Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *arXiv: 1602.02830* (2016).

[M C16b] M. Courbariaux, Y. Bengio and J. P. David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagation". In: *arXiv: 1511.00363* (2016).

[Mat19] Matplotlib. *Matplotlib*. 2019. URL: `https://matplotlib.org/` (visited on 2019-06-15).

[Mic15] Michael Nielsen. *Neural Networks and Deep Learning*. 2015. URL: `http://neuralnetwork-sanddeeplearning.com/index.html` (visited on 2018-12-12).

[Min14] Ming Zeng, Le T. Nguyen, Bo Yu, Ole J. Mengshoel, Jiang Zhu, Pang Wu, Joy Zhang. "Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors". In: *6th international conference on mobile computing, applications and services (MobiCASE)* (2014), pp. 197–205.

[Mob19] MobSemi. *MobSemi*. 2019. URL: `http://www.mobile-semiconductor.com` (visited on 2019-06-17).

[Nor18] Norman Jouppi, Cliff Young, Nishant Patil, David Patterson. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *IEEE Micro* 38 (2018-06), pp. 10–19.

[Nor19] Nordic. *Nordic*. 2019. URL: `https://www.nordicsemi.com/` (visited on 2019-06-17).

[Pyt19] Python. *Python*. 2019. URL: `https://www.python.org/` (visited on 2019-06-15).

[Qua18] Qualcomm. *Snapdragon 855 Mobile Platform*. 2018. URL: `https://www.qualcomm.com/products/snapdragon-855-mobile-platform` (visited on 2018-12-16).

[Que19] Questasim. *Questasim*. 2019. URL: `https://www.mentor.com/products/fv/questa/` (visited on 2019-06-15).

[Ren16] Renzo Andri, Lukas Cavigelli, Davide Rossi and Luca Benini. "YodaNN: An Architecture for Ultra-Low Power Binary-Weight CNN Acceleration". In: (2016).

[Ren18] Renzo Andri, Lukas Cavigelli, Davide Rossi and Luca Benini. "Hyperdrive: A Systolically Scalable Binary-Weight CNN Inference Engine for mW IoT End-Nodes". In: (2018).

[Sag17] Sagar Sharma. *Activation Functions: Neural Networks*. 2017. URL: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6` (visited on 2018-12-12).

[Shy17] Shyamal Patel, Johanna Pingel. *Introduction to Deep Learning: What Are Convolutional Neural Networks?* 2017. URL: `https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html` (visited on 2018-11-11).

[Soj16] Sojeong Ha, Seungjin Choi. "Convolutional Neural Networks for Human Activity Recognition using Multiple Accelerometer and Gyroscope Sensors". In: *International Joint Conference on Neural Networks* (2016), pp. 381–388.

[Son15] Song Han, Huizi Mao, William Dally. "Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *International conference on Learning Representations 2016* (2015).

[Son16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, William Dally. "EIE: Efficient inference engine on compressed deep neural network". In: *IEEE Annual international symposium on computer atchitecture* 43 (2016-06).

[Spy19] Spyglass. *Spyglass*. 2019. URL: `https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-power.html` (visited on 2019-06-15).

[Ste14] Stefan Duffner, Samuel Berlemont, Gregoire Lefebvre, Christophe Garcia. "3D gesture classification with convolutional neural networks". In: *IEEE International conference on Acoustic, Speech and Signal Processing (ICASSP)* (2014), pp. 5432–5436.

[Syn19] Synopsys. *Synopsys*. 2019. URL: `https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html` (visited on 2019-06-15).

[Ten19] Tensorflow. *Tensorflow*. 2019. URL: `https://www.tensorflow.org/` (visited on 2019-06-15).

[Viv17] Vivienne Sze, Yu-Hsin Chen, Joel Emer and Tien-Ju Yang. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* (2017).

[Xia16] Xiang Zhang, Yann LeCun. "Text Understanding from scratch". In: *arXiv:1502.01710v5* (2016-04).

[YuH17a] Yu-Hsin Chen, Joel Emer and Vivienne Sze. "Hardware for Machine Learning: Challenge and Opportunities". In: *arXiv 1612.07625v5* (2017).

[YuH17b] Yu-Hsin Chen, Tushar Krishna, Joel Emer and Vivienne Sze. "Eyeriss: An energy efficient reconfigurable accelerator for deep convolutional networks". In: *IEEE journal of solid-state circuits* 52.1 (2017-01), pp. 127–138.

[YuH18a] Yu-Hsin Chen, Joel Emer Vivienne Sze. "Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks". In: *arXiv 1807.07928v1* (2018).

[YuH18b] Yu-Hsin Chen, Tien-Ju Yang and Joel S. Emer. "Understanding the Limitations of Existing Energy-Efficient Design Approaches for Deep Neural Networks". In: 2018.

Appendix A

Supplementary Material

A.1 2D CNN structure

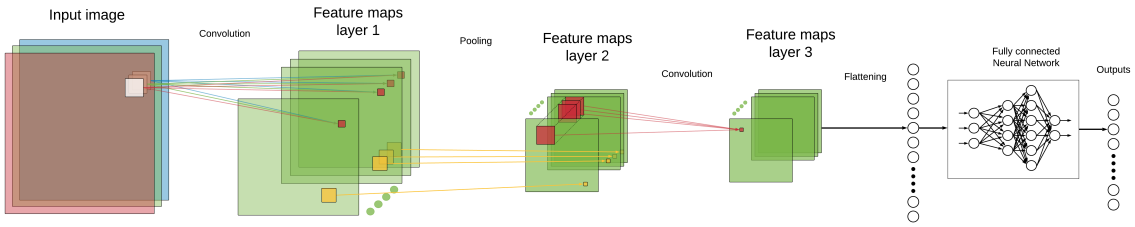


Figure A.1: A full CNN. Comparable to figure 2.4.

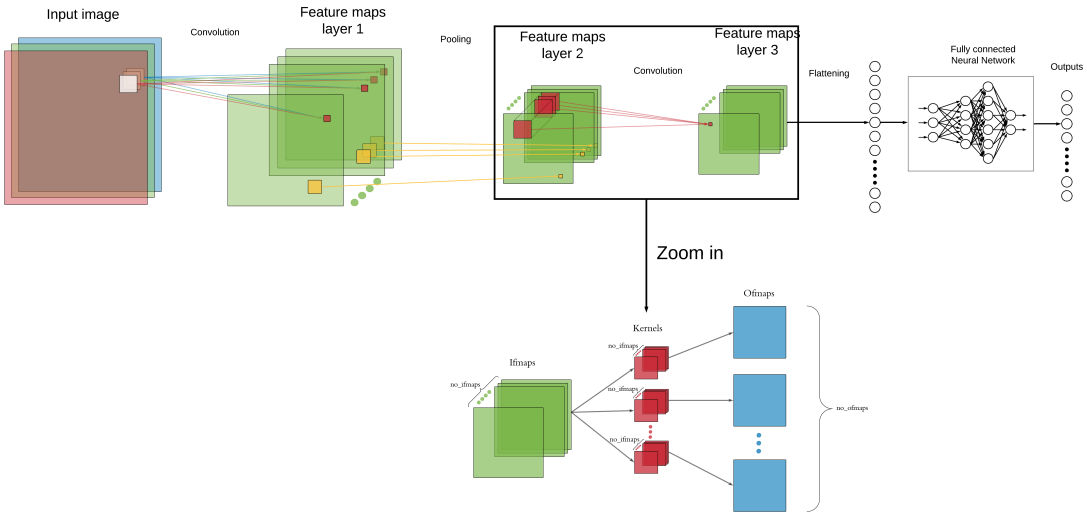


Figure A.2: The full CNN with one layer shown in more detail.

A.2 Timing diagram of the 1D CNN Accelerator

Table A.1: Timing diagram for the 1D CNN Accelerator with no ifmaps = 10, no ofmaps = 20, kernel size = 3, $K = 5$, $M \geq 20$

cycle	1	2	3	4	5	6	7	...	63	64
State	IDLE	INIT	LOAD KER							LOAD-
data in			$k_{0,0}[0]$	$k_{0,0}[1]$	$k_{0,0}[2]$	$k_{0,1}[0]$	$k_{0,1}[1]$	...	$k_{0,19}[2]$	b_0
ker values										
ifm value										

continuation ...

cycle	65	...	84	85	86	87	88	89	90	91
State	BIASES			LOAD IFMAP					COMPUTE	
data in	b_1	...	b_{19}	$i_0[0]$	$i_0[1]$	$i_0[2]$	$i_0[3]$	$i_0[4]$		
ker values									$k_{0,all}[0]$	$k_{0,all}[1]$
ifm value									$i_0[0]$	$i_0[1]$

continuation ...

cycle	92	93	94	95	96	97	98	99	100	...
State		IFM	COMPUTE			IFM	COMPUTE			...
data in		$i_0[5]$				$i_0[6]$				...
ker values	$k_{0,all}[2]$		$k_{0,all}[0]$	$k_{0,all}[1]$	$k_{0,all}[2]$		$k_{0,all}[0]$	$k_{0,all}[1]$	$k_{0,all}[2]$	...
ifm value	$i_0[2]$		$i_0[1]$	$i_0[2]$	$i_0[3]$		$i_0[2]$	$i_0[3]$	$i_0[4]$	...

continuation ...

cycle	389	390	391	392	393	394	395	396	397	398
State	IFM	COMPUTE								
data in	$i_0[79]$									
ker values		$k_{0,all}[0]$	$k_{0,all}[1]$	$k_{0,all}[2]$	$k_{0,all}[0]$	$k_{0,all}[1]$	$k_{0,all}[2]$	$k_{0,all}[0]$	$k_{0,all}[1]$	$k_{0,all}[2]$
ifm value		$i_0[75]$	$i_0[76]$	$i_0[77]$	$i_0[76]$	$i_0[77]$	$i_0[78]$	$i_0[77]$	$i_0[78]$	$i_0[79]$

continuation ...

cycle	399	400	401	402	403	404	405	406	407	408
State	IDLE	LOAD KER							LOAD IFMAP	
data in		$k_{1,0}[0]$	$k_{1,0}[1]$	$k_{1,0}[2]$	$k_{1,1}[0]$	$k_{1,1}[1]$	...	$k_{1,19}[2]$	$i_1[0]$	$i_1[1]$
ker values										
ifm value										

continuation ...

cycle	409	410	411	412	413	414	415	416	417	418
State	LOAD IFMAP			COMPUTE			IFM	COMPUTE		
data in	$i_1[2]$	$i_1[3]$	$i_1[4]$				$i_1[5]$			
ker values				$k_{1,all}[0]$	$k_{1,all}[1]$	$k_{1,all}[2]$		$k_{1,all}[0]$	$k_{1,all}[1]$	$k_{1,all}[2]$
ifm value				$i_1[0]$	$i_1[1]$	$i_1[2]$		$i_1[1]$	$i_1[2]$	$i_1[3]$

continuation ...

cycle	419	...	3287	3288	3289	3290	3291	3292	3293	3294
State	IFM	...	IFM	COMPUTE						
data in	$i_1[6]$	...	$i_9[79]$							

ker values		...		$k_{9,all}[0]$	$k_{9,all}[1]$	$k_{9,all}[2]$	$k_{9,all}[0]$	$k_{9,all}[1]$	$k_{9,all}[2]$	$k_{9,all}[0]$
ifm value		...		$i_9[75]$	$i_9[76]$	$i_9[77]$	$i_9[76]$	$i_9[77]$	$i_9[78]$	$i_9[77]$

continuation ...

cycle	3295	3296	3297	3298	3299	...	4856	4857	...	...
State	COMPUTE		STREAM OUT					IDLE	...	...
data out			$o_0[0]$	$o_0[1]$	$o_0[2]$	...	$o_{19}[77]$		...	...
ker values	$k_{9,all}[1]$	$k_{9,all}[2]$				...			...	...
ifm value	$i_9[78]$	$i_9[79]$				...			...	...

Where $k_{x,y}[n]$ means weight number n in the kernel used between ifmap x and ofmap y, b_y is the bias of ofmap y, $i_x[n]$ is sample number n of ifmap x, $o_y[n]$ is sample number n of ofmap y. *State* refers to the current state of the 1D CNN accelerator's main FSM. *data in*/*data out* is a merged version of all the different inputs and outputs that the accelerator has. The timing diagram is simplified slightly, e.g. some simple states for resetting counters are omitted.

A.3 Area plotted in 3D

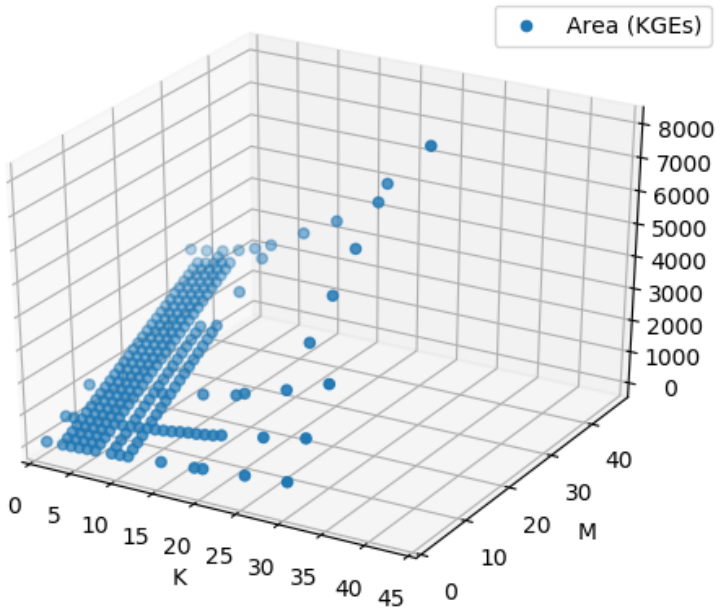


Figure A.3: Scatter plot of area as a funtion of K and M. $B = 90$ and RAMs are 8k

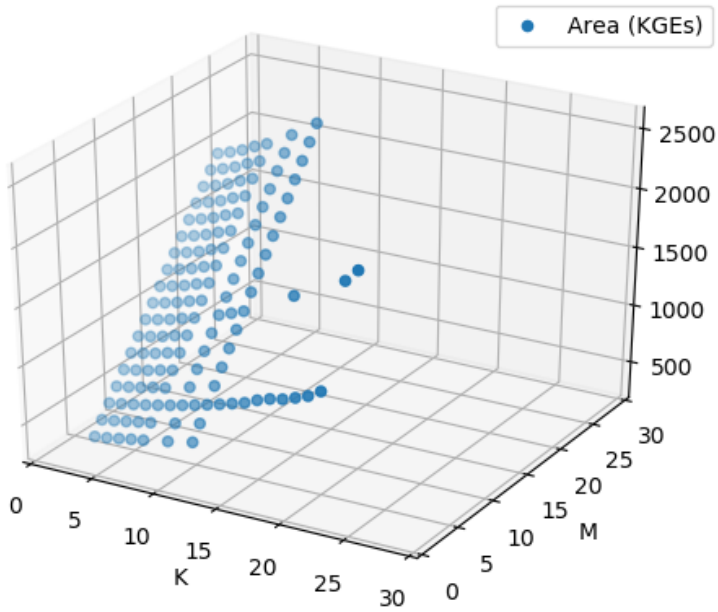


Figure A.4: Area as a funtion of K and M. Same as A.3 Zoomed in somewhat.

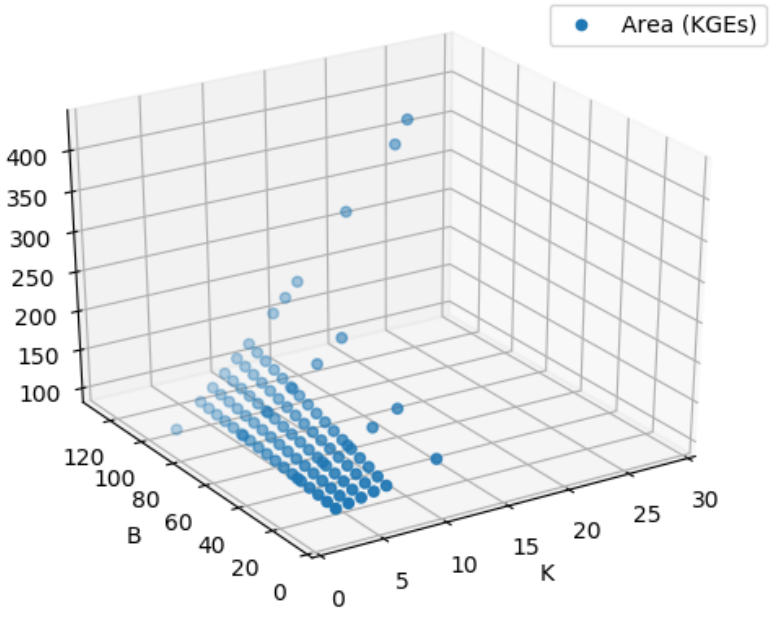


Figure A.5: Area as a funtion of K and B. $M = 1$ and RAMs are 8k