Near-Memory Computing Compiler for Neural Network Architectures

ALEX ALLFJORD
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Near-Memory Computing Compiler for Neural Network Architectures

Alex Allfjord
alex.allfjord@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Joachim Rodrigues

Examiner: Pietro Andreani

February 21, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

With an increased popularity of machine learning, both higher performance and more energy-efficient circuits are needed to meet the demands of increasing workloads. This master's thesis focuses on convolutional neural networks and implements a compiler that generates an accelerator architecture that can be tailored to performance needs. The implemented architecture utilizes near-memory computing to gain increased performance and higher energy efficiency. This report gives an overview of the implemented architecture. Area and performance results for an example use-case are presented and ideas for future improvements are listed.

Popular Science Summary

Speeding up the Development of Machine Learning Accelerators

The world has in the last decade seen an increased use of machine learning and computer vision. With this comes a need for increased performance, lower energy usage, and shorter development times. This thesis provides a stepping stone in the right direction to achieve this.

Machine learning is an ever more popular way to let computers or computer models learn and perform complex tasks. An application area closely coupled with machine learning is computer vision. Computer vision involves letting computer models see visual information (like images and video), by interpreting and extracting information out of a given input. One popular area that can perform such a task is convolutional neural networks.

These convolutional neural networks are structured in layers and process a set of input data. The data moves through the layers, and various mathematical operations are performed on the data. At the output, information or a conclusion about the data is provided. One of the layers is the convolutional layer, this layer "moves" a filter over the input. The filter itself is a set of predetermined weights that multiplies with the input. These multiplications are then summed. The subsequent result can then be handed over to the next layer. For increased performance, the convolutional operations can be performed by a hardware accelerator specialized for this purpose.

A hardware accelerator is separate from a computer's processing unit. Placed near the computer's memory it can be called near-memory computing. Being physically closer to memory has two advantages: less energy is required to transfer the data and the data transfer can be faster. Memories are often made up of many smaller memories, called "macros". Placing an accelerator near the memories allows us to connect to all the individual macros inside the memory. This has the advantage of accessing many parts of memory at once and increasing data throughput, letting us process more data if we have the resources to compute it.

Designing a hardware accelerator is a complex and time-consuming task, as it needs to be implemented, tested for functionality in simulation, and integrated into a bigger system. When everything is working properly it can be translated

into a physical design and manufactured.

Involving the described areas above, the produced thesis aims to speed up the design of hardware accelerators for convolutional neural networks. Providing a compiler that takes a set of input parameters and tailors an accelerator architecture from these parameters allows users to choose the performance of the accelerator, fast and easily integrate it, and test it. Computations performed by the accelerator are the convolutional operation/filter operations mentioned. The architecture is envisioned to be placed near memory by the user to take advantage of near-memory computing, increasing energy efficiency and performance. A use case with statistics of an example generated with the compiler is also provided.

Acknowledgements

I want to express my gratitude to my supervisors for the opportunity and help with this thesis. You have given me much freedom with this work which has made it a lot of fun and I thank you for that. I also want to take this opportunity to say thank you to my partner for always supporting me no matter what.

Acronyms

ASIC Application Specific Integrated Circuit
AW Address Width
BBA Bias Base Address
CIM Compute In Memory
CNN Convolutional Neural Network
CPU Central Processing Unit
DW Data Width
FBA Filter Base Address
FRA Filter Result Accumulator
FS Filter Size
FSO Filter Sum Out
HDL Hardware Description Language
HW Hardware
IBA Input Base Address
ID Input Depth
IP Intellectual Property
IS Input Size
MAC Multiply Accumulate
MFS Maximum Filter Size
MID Maximum Input Depth
MIS Maximum Input Size
ML Machine Learning
MNF Maximum Number of Filters
MS Maximum Stride
NF Number of Filters
NMC Near-Memory Computing
NN Neural Network
PF Parallel Filters
PD Parallel Depth
PDK Process Design Kit
PI Pick Input
RD Read Depth

RDI Row Data In
ReLU Rectified Linear Unit
RRO Row Result Out
RSA Result Save Address
RTL Register-Transfer Level
SSD Solid-State Drive
TSB Truncation Start Bit
VHDL Very High Speed Integrated Circuit Hardware Description Language
WPC Weights Per Clock
WDI Weight Data In

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Machine learning (ML) has seen increased use in recent years thanks to the improved performance of circuits and the availability of big data sets [1]. A popular ML technique is the use of neural networks (NNs). Neural networks are used for many different tasks such as classification, pattern recognition, filtering, data compression, regression analysis, etc. The use of NNs is both computationally heavy and memory-intensive.
Computations on a classical central processing unit (CPU) architecture, like the Harvard or Von Neumann, are inefficient with memory being the main bottleneck resulting in slow data transfers. A possible technique is to develop a dedicated ML accelerator on application specific integrated circuits (ASICs) for improved energy efficiency and performance. The complexity of developing hardware (HW) coupled with that different NNs have different requirements. Thus, designing, developing, and testing new accelerators is costly and time-consuming. While dedicated ASICs are a promising solution for the problem a way of simplifying or generalizing the development is needed.
NNs are memory intensive, and a lot of performance and power is lost in the memory accesses and memory transfer of data [2]. One approach to improve this is placing the logic and memory near each other, called near-memory computing (NMC) allowing for a great reduction of power from data transfers and a decrease in latency. To aid in the development of NMC accelerators for ML this master's thesis details a technique to generate a HW description of a near-memory ML architecture easily and automatically, by giving just a few input parameters. This will speed up the process of designing and testing an architecture with different parameters for optimal performance given a particular problem while taking advantage of the benefits of NMC.

1.1 Background and Motivation

With the rising popularity of ML usage, the need for energy-efficient implementations has increased as well. Autonomous cars, augmented reality, voice automation, computer vision, etc., benefit from the use of ML. These computing areas require high-performance HW, and would benefit greatly from power-efficient HW since available energy (battery-operated) usually is limited.
In NNs, neurons perform computations and are integrated by interconnected lay-

ers. The output from a neuron is a weighted sum of its inputs, which are the outputs of the previous layer. This process is realized by multiplication and accumulation (MAC) operations, and poses a memory-heavy task, with inputs and weight loaded from memory and the result stored back in memory. The facts, coupled with the problem of the increasing latency gap between processors and memory, the memory/bandwidth wall [3] has made traditional CPU-based architecture inadequate to handle the memory-heavy tasks required for NNs.
Together with the rise of ML, NMC has also seen an increase in popularity. With technological advancement it is now possible, for example, to use solid-state drives (SSD) with computational capabilities for data processing near memory [1]. While other approaches exist, such as compute in memory (CIM), this approach introduces limitations in what computations can be implemented and requires changing the structure of current already optimized memories, resulting in increased power usage [4]. The NMC approach enables good energy efficiency without changing the already efficient memories while offering increased performance in memory-intensive computations, and for that reason, it plays an important part of the accelerator design in this work.

1.2 Project Aim

The goal of this master's thesis is to create a platform to enable the auto-generation of near-memory NN accelerator HW from a set of input parameters for configuration. The parameters allow the HW to be tailored to the performance needs of the HW user.

1.3 Report Disposition

The report is structured in the following way:

Chapter 2, Background Theory gives a short overview of the theoretical knowledge needed to understand the developed architecture.

Chapter 3, Accelerator Architecture introduces the compiler parameters and shows the basic structure and components of the compiler generated architecture.

Chapter 4, Compiler Generated Example shows performance statistics for some synthesized versions of the compiler generated HW description and a discussion about the generated results.

Chapter 5, Conclusion concludes the project and provides some examples of improvements that could be carried out in the future.

Chapter 2
Background Theory

This chapter introduces the basic theory which is relevant for the understanding of this project and the developed architecture.

2.1 Near-Memory Computing

NMC is an architecture type that places the computational elements and memories physically close to each other [5]. This helps to reduce latency, improve throughput bottlenecks, and increase energy efficiency. In a traditional architecture, data transfers to and from memory utilize a bus. This increases energy usage, latency and reduces the bandwidth. Memories utilizing a bus for data transfers see a reduction in the possible throughput since the bus might be utilized by multiple entities. Connecting computational HW directly to the macros inside the memory allows the usage of the full throughput of the macros, while at the same time eliminating the extra power inefficiency of using a bus. This project provides an architecture that takes advantage of the NMC benefits, such as increased bandwidth. This is achieved by connecting directly to the memory macros and by operating mostly independently from the main CPU, further reducing bus accesses to achieve increased energy efficiency.

2.2 Hardware Compiler

For this work, a HW compiler is defined as a program or script that takes parameters and constraints to create a HW architecture accordingly. The HW is described by a Hardware Description Language (HDL). This is what the compiler of this project produces, a set of HDL files that needs interpretation by some synthesis tool to be translated into a physical implementation. The current compiler implementation takes a set of parameters to set up constraints used in determining the size of the architecture to implement, which in turn translates to a performance at a specific area cost.

2.2.1 HDL

The compiler is written in VHDL which stands for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. VHDL is a popular HDL used

to describe basic components of digital systems and their connections in a coding language style. This HDL code is then used to translate the described functions of the components in the code to physical components that replicate the behaviour the code describes. A VHDL reference manual is found in [6].

Register Transfer Diagram

The VHDL code enables a translation into register transfer level (RTL) diagrams. The RTL diagram shows components like registers, multiplexers, adders, etc. The architecture given by the compiler is presented in chapter 3. Simplified RTL diagrams are shown to help explain the architecture.

2.2.2 Synthesis

The synthesis process takes the HDL code produced by the compiler and converts the code to a HW structure. Generally, the HDL code is translated to a set of generic gates, and then mapped to specific HW gates or blocks. The HW blocks are read from a library, provided from a process design kit (PDK) provider. This means that depending on the PDK used, the physical implementation of the circuit differs. The synthesis tool used to generate the results in chapter 4 is Genus Synthesis Solution by Cadence [7].

Power Simulation

Various tools exist to estimate circuit parameters such as power consumption and timing information. To estimate power consumption for a synthesized design, PrimeTime by Synopsys was used [9]. PrimeTime takes library information provided by a technology manufacturer together with variables such as supply voltage and operating temperature. Switching information from a simulation of the circuit is then provided to PrimeTime to help estimate the power consumption of the design.

2.3 Neural Networks

The basic building block of a NN is the neuron. The neuron takes a dot product between a set of inputs and a set of predetermined weights with the addition of a predetermined bias value, this result is the input to the function f where f is some non-linear function called an activation function [10]. The general function of a neuron is defined as

$$f([in_1, in_2, \dots in_n] \cdot [w_1, w_2 \dots w_n] + bias) = y. \tag{2.1}$$

Putting neurons in a layer where the output of the layer is the input to another layer, the general structure of Figure 2.1 is obtained. In the figure, the input layer sees an input vector represented by numbers. This could be an image, outputs from another network, etc. At the output layer a vector of numbers representing information or a conclusion about the input are found. This layer structure in the network gives rise to (2.2), which can be simplified to (2.3). The resulting output

Y in (2.3) is then fed to the activation function and the results are passed to the next layer. By adjusting all individual weights and biases of the neurons, using the right type of activation function, using a certain number of neurons in a layer, and using a certain number of layers, the network is tuned to produce the desired outputs for some set of inputs.

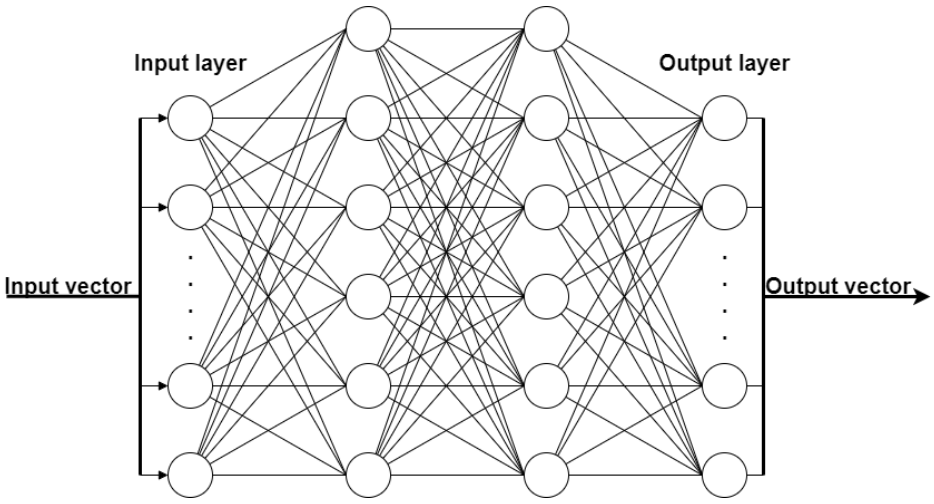

Figure 2.1: A basic neural network example with circles representing neurons and lines representing weighted connections. The input layer takes a vector of numbers and the output layer outputs a vector representing information or a conclusion about the input.

$$\begin{bmatrix} in_{11} & in_{12} & \dots & in_{1c} \\ in_{21} & in_{22} & \dots & in_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ in_{r1} & in_{r2} & \dots & in_{rc} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1r} \\ w_{21} & w_{22} & \dots & w_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ w_{c1} & w_{c2} & \dots & w_{cr} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_r \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix} \tag{2.2}$$

$$IW + B = Y \tag{2.3}$$

2.4 Convolutional Neural Networks

Many classes of NNs exist, and the NN example in the previous section is selected as a general example of the feed-forward type of network. A more complex type of network in the same class is convolutional neural networks (CNNs). This network applies a filter window that performs the convolutional operation over the input layers. This makes this type of network good for image processing since it helps to extract and break down individual features that might be in the input [11].

2.4.1 Layers in a CNN

To construct a basic CNN, different layers are used, some examples include:

- Convolutional layer
- Pooling layer
- Fully connected layer

Different kinds of layers exist, but these three are some of the most commonly seen building blocks. The convolutional layer contains the aforementioned filters that move over the input. The pooling layer is usually placed after the convolutional layer. The pooling layer's task is to reduce the size of the convolutional layer's output. Two popular ways to implement this are maximum pooling and average pooling. To accomplish this a partial region of the input to the pooling layer is taken, and then either the maximum value or the average value of the region is forwarded to the next layer. This is performed over the whole input to the pooling layer, effectively reducing the size of the input. The fully connected layer has the structure of one layer seen in Figure 2.1. If the two-dimensional or three-dimensional output from a convolutional or pooling layer is "flattened", converted to a one-dimensional vector makes inputting it to the fully connected layer possible. Although a CNN accelerator architecture preferably implements all the mentioned layers to enable computation of a complete network. Due to time constraints and the complexity of creating an architecture that is flexible and scalable, only the convolutional layer has been the focus of this work. The convolutional layer usually (depending on the CNN implementation and varies from case to case) contains a large proportion of the total multiplications in the CNN. Therefore, it is chosen as a good starting point for the compiler to implement.

2.4.2 Convolution Layer

The basic operation of the convolutional layer is described as

$$X(f,r,c) = F\left(B_f + \sum_{d=0}^{D-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1} I_{d,(r+n),(c+m)} * W_{f,d,n,m}\right), \qquad (2.4)$$

where f represents the filter to be computed, r represents the row of the output and c represents the column of the output. The depth of the filter is represented by D. The row and column size of the filter are represented by N and M, respectively. The input is represented by I where the first index d is the depth. The second index, $(r+n)$, is the row. The third index, $(c+m)$, is the column. The weights of the filter are represented by W, with the first and second index, f, and d, representing the filter to be computed and its depth respectively. The third and fourth indexes represent the row and column of the filter respectively. To the sum, a bias value B_f is added to provide an offset of a filter result. Finally, the complete sum is the input of the activation function F to provide non-linearity to the result. The layer operation is called a two-dimensional convolution, even though the input and filter are in three dimensions, row, column, and depth. The filter only "moves" over the row and column since the filter depth and input depth are the same, and

the full depth is summed for every computed output X. Applying the equation for all rows, R, and columns, C, produces a two-dimensional output per filter f. Since one layer contains multiple filters, K. The final output from the layer will have the size of R by C by K where K is the depth of the output also called the number of output channels.

2.4.3 Activation Function

Many different types of activation functions exist. Some examples include the functions seen in (2.5) through (2.7).

$$\text{Sigmoid: } f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

$$\text{Tanh: } f(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.6}$$

$$\text{ReLU: } f(x) = \max(0, x) \tag{2.7}$$

One of the most commonly used is Equation (2.7), ReLU (Rectified Linear Unit), [12]. $Max(0, x)$ simply translates to a function that is zero for negative values of x and is equal to x for positive x. This relatively simplistic function is also simple to implement in HW. Since it only requires the designer to look at the sign of the result x, and output x if positive otherwise output a zero. For these reasons, this activation function was chosen to be implemented by the compiler.

2.4.4 Quantization

Floating point computations are used for multiplications and additions that contain decimals. While floating points do give a higher precision, they require more HW, are slower to compute and are more complex to implement. An alternative is to implement all computations in fixed integer multiplication, [13]. This reduces accuracy but enables each computation to be computed in one clock cycle. First, a certain bit precision has to be chosen, for example, N bits. This gives a range of -2^{N-1} to $2^{N-1} - 1$. Now the maximum floating point value and the minimum floating point value found in the NN to be computed is mapped to the highest and lowest integer range value, respectively. All other values in the NN now fall somewhere in between the maximum and minimum ranges and are to be mapped linearly to a corresponding integer. However, the values do not necessarily need to correspond to integer values. Using fixed point arithmetic, some bits are representing integers and some bits are representing the fractional part. An example of fixed point number representation is provided in (2.8). It is shown that fixed point representation can be described by an integer and a multiplication by two to some negative power. This implies that the multiplication between two fixed point numbers can be carried out as integer multiplications with the result shifted with $2^{-(m+n)}$, where m and n are the number of bits used for the fractional part in each respective multiplicand.

$$
\begin{aligned}
&101.11001 \equiv \\
&1 * -(2^2) + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 0 * 2^{-4} + 1 * 2^{-5} \\
&= (1 * -(2^7) + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0) * 2^{-5} \\
&= -71 * 2^{-5} \\
&= -2,21875
\end{aligned} \tag{2.8}
$$

Therefore, in HW, the multiplication is carried out in the same way no matter if integer or fixed point representation is used. If fixed point representation is needed, the user of the system needs to interpret the generated results correctly. This means that knowledge of where the radix point ends up after multiplication is needed. If the radix point is in position n for one multiplicand and in position m for the other multiplicand, the result will have its radix point in position $n+m$.

2.4.5 Truncation

As multiplications and additions are carried out, more and more bits are needed to represent the growing result. When computing the subsequent layer in the CNN the same multipliers need to be used, i.e., the same number of bits as the previous layer is needed for representation. One way to do this is to truncate the results back to the original input precision [14]. The act of truncating entails discarding the lower-valued bits and to keep a subset of the bits from the upper range. This will lower the precision of the system with the trade-off that the same-sized multipliers are used to carry out the needed multiplications for all CNN layers. Therefore, a user of the architecture needs to investigate the optimal point to truncate for each layer before using the architecture.

Chapter 3
Accelerator Architecture

This chapter will introduce the compiler parameters together with RTL diagrams to give an overview of the developed architecture.

3.1 Compiler Parameters

A short overview of the compiler parameters and their ranges will be presented.

3.1.1 Hardware Parameters

The different memories the architecture requires are given in Table 3.1. All the names represent one memory that needs to be connected to the generated architecture, except Buffer. The reason for this is that multiple buffer memories are needed, the number of needed buffer memories depends on the parallel filter (PF) parameter, which will be given later. Each memory is given by two parameters that need specification, these parameters exist so that the architecture can interact with the given memories correctly. These are **data width (DW)** and **address width (AW)**. The parameters take integer values that represent the bit array length of a given element. Their ranges for DW and AW are to be anything from one and upward with no restrictions from the compiler side. Different memory intellectual properties (IPs) will come with different DW and AW with their own restrictions. The memory parameters give the user the control and flexibility needed to create and connect the HW to different-sized memory systems that fit the performance needed by the user.

Table 3.1: Names for all the needed memories.

Required memories
Input
Weight
Bias
Output
Buffer

One restriction of the memory parameters is that the **bias memory DW** and **buffer memory DW** currently need to have the same value. This is due to the fact that the number of bits for these two memories controls the number of bits used for the final accumulated results. Thus, the sizes of these two vectors need to be big enough to ensure there is no overflow after accumulating all results of a convolution operation. Most likely, **input memory DW** and **output memory DW** will also have the same value if the user wants to be able to switch between them for input and output. The addition needed for this will be shown in section 3.2. All memory parameters; **input/weight/bias/output/buffer memory AW**s are required to specify the number of bits needed to interact with the memories and to limit counters in the architecture to not use more bits than needed to index the specified address range. The user needs to make sure that memories with an address range that is big enough for the input data, results, and weights to fit in their respective memories.

The compiler parameters given in Table 3.2, set the maximum sizes of the convolutional layers that will be computable, define precision and determine parallelization/performance. All values are specified as integer values.

Table 3.2: Hardware parameters to specify compiler configuration.

Accelerator parameters	Acronym
Data Width	DW
Maximum Filter Size	MFS
Maximum Input Size	MIS
Maximum Input Depth	MID
Maximum Number of filters	MNF
Maximum Stride	MS
Parallel Filters	PF
Parallel Depth	PD

Data Width is used to specify the number of bits used for each input, weight, and output value. All of these values are treated as a two's complement numbers and will be referred to as a "data word". DW is specified as any positive integer but, when multiplied with another positive integer N, the following needs to apply:

$$DW * N \leq M \tag{3.1}$$

and

$$M \ mod \ (DW * N) = 0 \tag{3.2}$$

where M is any of the integer widths specified for **memory DW** of the memories in Table 3.1. This ensures that an integer number of words fits in the specified memories and completely fills one address row with values.

Maximum Filter Size (MFS) sets the maximum possible filter size that can be used for a convolution operation. The set size is squared such that an

integer 3 specifies a 3 by 3 filter and an integer 9 specifies a 9 by 9 filter. The minimum allowed value is 3 and is incremented in steps of 2 to any desired size. The depth of the filter always follows the depth of the input.

Maximum Input Size (MIS) specifies the maximum two-dimensional size of an input to a layer. That is, if MIS is given as I the maximum allowed input is I by I data words in size. The minimum value of **MIS** is three following the minimum filter size.

Maximum Input Depth (MID) refers to the maximum size of the third dimension of the input, usually referred to as input depth or the number of input channels, and takes any positive integer value. With **MIS** given by I and **MID** given by D the maximum allowed size of an input is I by I by D.

Maximum Number of Filters (MNF) sets the maximum allowed number of filters/the number of output channels for a given layer. It is set to any integer value starting at one.

Maximum Stride (MS) sets the maximum allowed stride for any given layer. It is set to any integer value starting at one.

Parallel Filters (PFs) set how many filters of one layer that the architecture calculates in parallel. The minimum integer value is one and the maximum allowed value depends on **output memory DW** / **DW**.

Parallel Depth (PD) sets how much of the input depth of a layer the architecture calculates in parallel. The minimum integer value is one and the maximum allowed value depends on **input memory DW** / **DW**.

3.1.2 Run Time Parameters

Run time parameters are the specifications for a specific layer that the user wants to compute. Compared to the previous HW parameters, the run time parameters do not influence the architecture data path or the HW that is generated after synthesis. The run time parameters have their ranges limited by the HW parameters. The run time parameters all take an integer as input and are summarized in Table 3.3.

Input Size (IS) is limited by **MIS** as its maximum value, the minimum value is the minimum filter size, three. If the specified **IS** is I for some layer the architecture expects the input for this layer to be I by I words in size.

Input Depth (ID) specifies the depth/number of channels in the input of the current layer. The maximum value is specified by **MID** and the minimum value is one.

Filter Size (FS) is the two-dimensional size of a filter that convolves over the input. If the given **FS** is F, the filter will have a size of F by F with the same depth of the input. The maximum value for **FS** is **MFS**, and the minimum size is the minimum **FS**, three. As **MFS** is specified in increments of two, so is **FS**, starting at the minimum value.

Table 3.3: Run time parameters for convolutional layers and their ranges.

Run time parameters	Acronym	Min.	Max.
Input Size	IS	3	MIS
Input Depth	ID	1	MID
Filter Size	FS	3	MFS
Stide	-	1	MS
Padding	-	0	1
Number of Filters	NF	1	MNF
Truncation Start Bit	TSB	DW	Bias memory DW
Relu	-	0	1
Filter Base Address	FBA	0	$2^{(Filter\ memory\ AW)} - 1$
Bias Base Address	BBA	0	$2^{(Bias\ memory\ AW)} - 1$
Result Save Address	RSA	0	$2^{(Result\ memory\ AW)} - 1$

Stride is the step size of the filter and is the same in both two-dimensional directions. The minimum value is 1 and takes any integer up to the set **MS**.

Padding is set with 1/0 indicating yes/no to use "same" padding. Same padding adds zeros around the input to ensure that the two-dimensional size of the output is the same as the input.

Number of Filters (NF) specifies the number of filters to run over the input, which translates to setting the output depth/number of output channels. The maximum allowed value is set by **MNF** and the minimum allowed value is one.

Truncation Start Bit (TSB) specifies from what bit index the results of a convolution will be truncated. The output will have the same word size as the inputs and weights determined by **DW**. The result vector before truncation is set to have the same number of bits as the **bias memory DW**. At some index, within this vector, a truncation point is set. Truncation from the set index point includes the start bit and **DW** number of bits under this point. An example: If **DW** equals eight and **bias memory DW** equals 32, the options for the **TSB** are 32 down to 8. Since **DW** is set to 8, the lowest bit truncation starts from is 8. If **TSB** equals 18 for example, the truncated result vector for the output results includes bit 18 down to (and including) bit 11.

Relu assumes a value of 1 or 0 indicating yes/no. If it assumes '1', relu will be applied to the output results and any negative value will be set to zero.

Filter Base Address (FBA) is to be set to the address which contains the first weight for a particular layer.

Bias Base Address (BBA) is to be set to the address which contains the first bias value for a particular layer.

Result Save Address (RSA) will be the address where the first results of a layer will be saved with the following results saved to subsequent addresses.

All these parameters are called run time parameters and need to be specified before synthesis. Further development of the compiler would allow these run time parameters to be given during operation. However, in the current version the parameters are saved as static read-only values and thus the parameters are not to be given or changed during operation. This is to decrease the number of parameters that are needed to be given to the HW for a computation. Based on the given run time parameters, many other values needed by the architecture compiler are pre-calculated. This limits the size of many control elements, thus reducing area and power, but also makes a trade-off with the architecture flexibility. To generalize the architecture to one that can be given the run time parameters at operation, the control logic needs to change to use bigger counters and bit vectors to be able to accommodate any unknown parameter configuration. If this change is made additional values need to be pre-calculated during operation for the HW to function properly. Provided in (3.3) is an example of one of these pre-calculated values. The example shows the needed calculation of a "jump" in memory address when loading different filters.

$$\frac{(FS)^2}{(\frac{weight\ memory\ DW}{DW})} \tag{3.3}$$

In (3.3), the "jump" in memory is calculated from the size of the current filter divided by how many data words fit into one address. This calculation contains both a square and two divisions. These operations are HW-expensive to implement. Changing the architecture into the more generalized version, these values that are more complex to calculate need to be provided at run time, and thus some added pre-calculation is needed. However, this will not affect run time performance since it is a one-time calculation. After calculation, the values are to be stored away, reused when needed again, and thus do not add any major overhead to the operation of the architecture.

3.2 Generated Architecture

RTL diagrams will be presented with the purpose of furthering the understanding of the presented compiler parameters and how they affect the performance of the architecture. The diagrams will mostly include the architecture data path. Most units presented have some control logic, however, the control logic does not change much with the given HW parameters and is therefore excluded to simplify the diagrams. At most, some counters change their upper limits, which means more bits might be needed to represent a number, but the operation of the logic mostly stays the same. Starting with the core of the architecture, the surrounding units will be built up and included until the whole architecture has been presented.

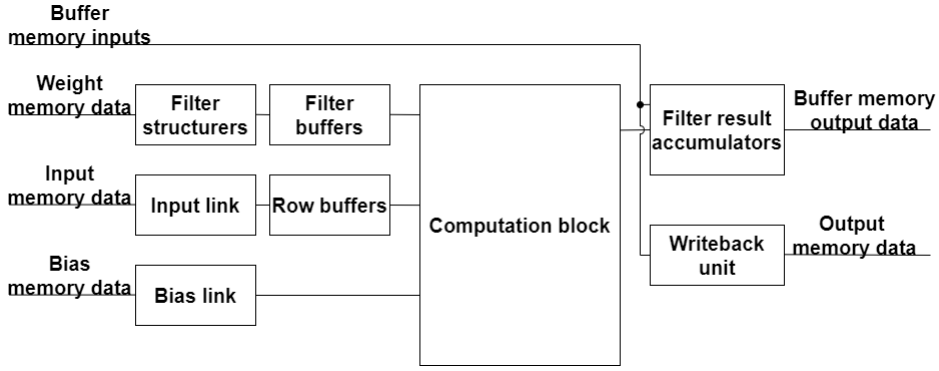

Figure 3.1: Architecture overview of module connections.

3.2.1 An Overview

For an initial overview, a basic diagram showing the modules to be explained in the chapter is shown in Figure 3.1. The diagram shown is simplified and does not contain control blocks and control signals to keep the diagram clear and understandable.

3.2.2 Basic Multiplier Block

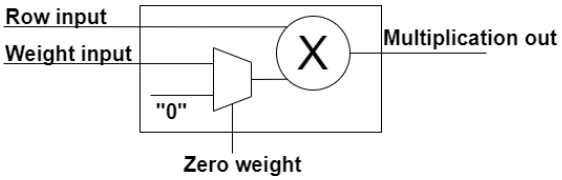

Figure 3.2: BM block.

The most basic building block is the multiplier shown in Figure 3.2. It consists of a multiplication of the two inputs, row data, and weight data. Both inputs will be **DW** number of bits in width. Multiplication will be signed resulting in the multiplication out signal being **DW** * 2 in width with one bit as a sign bit. The zero weight signal is a control signal that gives the option to force multiplication with a zero instead of the weight input effectively setting the multiplication output to zero.

3.2.3 Filter Row

The basic multiplier is used to implement a row of **MFS** number of multipliers seen in Figure 3.3, called a filter row. One of the multipliers' inputs is set to row data in, RDI, and the second connects to **MFS** number of registers that are chained together. Weights are inserted one by one from weight data in, WDI, and shift from one register to the next. When all weights of a filter row are loaded,

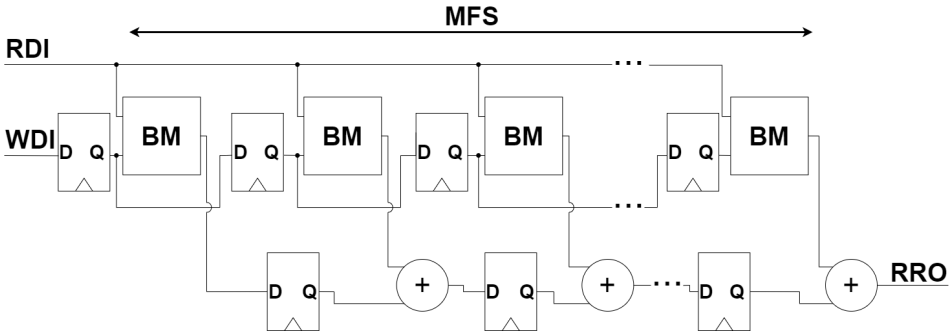

Figure 3.3: Row filter, BM denotes the basic multiplier. Computes one row of one filter over one row of inputs.

the input of row values starts. The multiplied result is added to the previous multipliers result and latched in a register. This means at the start the first row value is multiplied by the filter rows right most weight and the result is obtained at row result out RRO. In the next clock cycle, after the first result is obtained, the next row value is inputted and multiplied with the rightmost filter weight. This is added together with the first row value multiplied by the second rightmost filter weight. In the next clock cycle, the sum of three multiplications is obtained. This continues until the sum of all **MFS** number of multipliers is obtained. Thus, the filter row calculates the result of a filter moving over an input row, as described by a CNN filter convolution. The result from the multipliers is $\mathbf{DW} * 2$ number of bits. The subsequent and repeated addition means more bits are needed for the summed result. Therefore, the registers after the addition and the output have a bit array length of $2 * \mathbf{DW} + \log_2(\mathbf{MFS})$.

3.2.4 Filter Slice

The filter rows are arranged into a column as seen in Figure 3.4, which will be called a filter slice. The filter slice now consists of $\mathbf{MFS}^2$ number of multipliers and implements a square moving filter over its input. The filter rows are summed by adders to produce a single result at filter sum out (FSO) for a square filter. The output now has the bit array length of $2 * \mathbf{DW} + 2 * \log_2(\mathbf{MFS})$. To compute a filter smaller than $\mathbf{MFS}^2$, the "zero weight" signal is used in the multiplier block. For example, to compute a filter of size $(\mathbf{MFS} - 2)^2$ the two top rows of multipliers and two leftmost columns of multipliers are switched to zero.

3.2.5 Filter Core

Grouping multiple filter slices gives the filter core seen in Figure 3.5. The idea is to use one filter slice to compute values for one filter and another slice for another filter. This is because different filters share the same inputs allowing parallelization without being limited by the input memory bandwidth. The row inputs are the same as in Figure 3.4, but shown here as an array. Increasing the parallelization increases the number of weights that need to be loaded before computations begins,

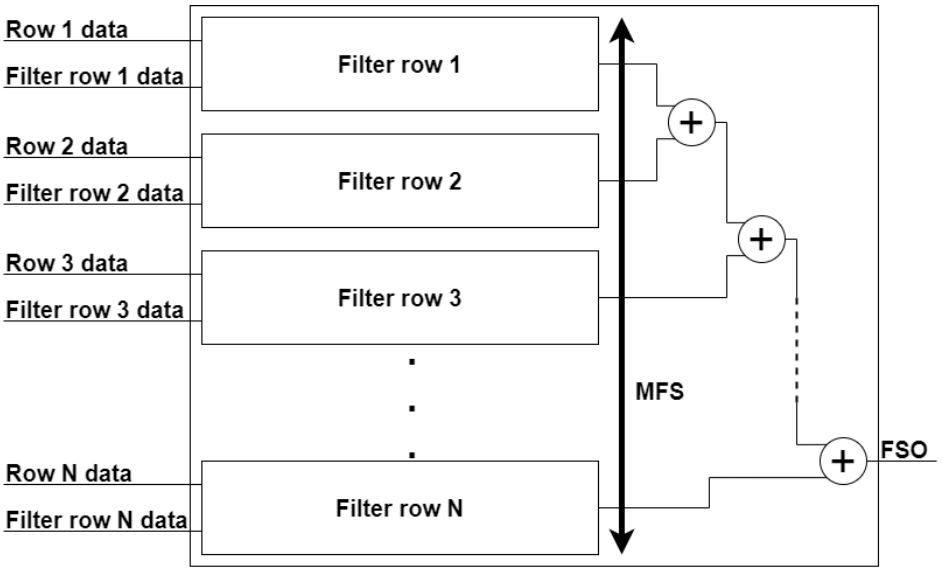

Figure 3.4: Filter slice, sums results of multiple filter rows.

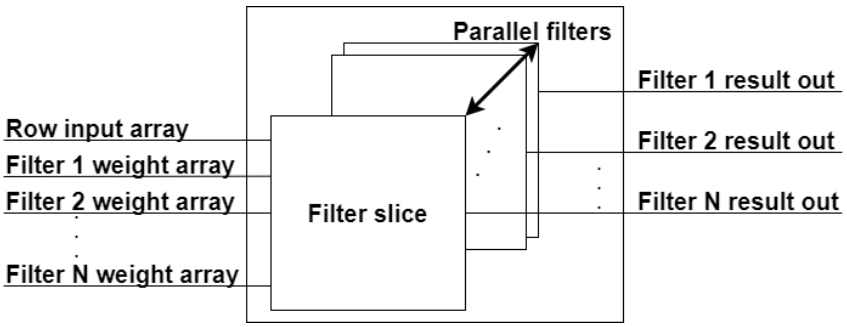

Figure 3.5: Filter core, groups together multiple filter slices.

which needs to be considered by the user. To control the number of slices in the filter core, the **PF** parameter is used. Also, observe that now there is **PF** number of outputs produced each clock cycle.

3.2.6 Computation Block

To further parallelize, assuming the bandwidth of the input memory is sufficient, multiple input row values are loaded from different depths of the input. Thus, using multiple filter cores as seen in Figure 3.6, enables different depths of filters to be calculated. To control the number of filter cores, i.e., the current depth of the input/filters that are worked on in parallel, the **PD** parameter is used. This new block also contains adders to sum the results from different depths, giving a total of **PF** number of outputs. The width of the output is now the previous bit array length added with $\log_2$ of the number of **PD** used: $2 * \mathbf{DW} + 2 * \log_2(\mathbf{MFS}) + \log_2(\mathbf{PD})$. The use of multiplexers before the input of each adder enables the option to take the results of a certain filter core, i.e., the result of a certain depth calculation, and zero it instead of adding it. The reason for this is best explained with an example: Calculating a filter with the depth of seven and **PD** set to four. This means that after the first four inputs of the depth are calculated, only three in depth are left of the input/filter to calculate, and thus the results of the fourth filter core need to be set to zero. Therefore, multiplexers switch the results from the fourth filter core to zero.

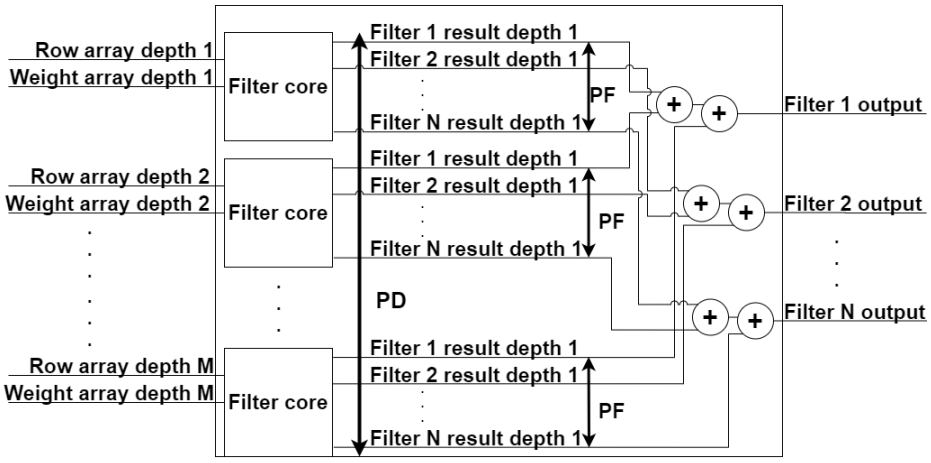

Figure 3.6: Computation block, that sums the results from the filter cores.

3.2.7 Weight Loading

In order to load the weights of a filter, buffers are used. The buffers allow the loading of weights from a memory while computations in the computation block are still ongoing. The loaded weight then only needs to move from the buffers into

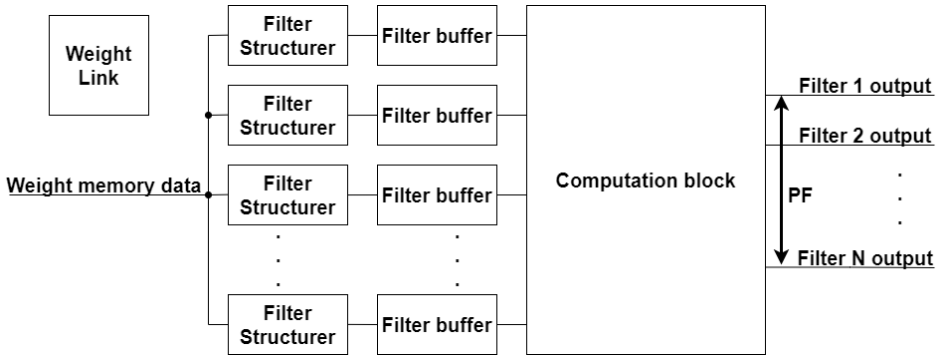

Figure 3.7: Overview of the different weight management blocks and their connections.

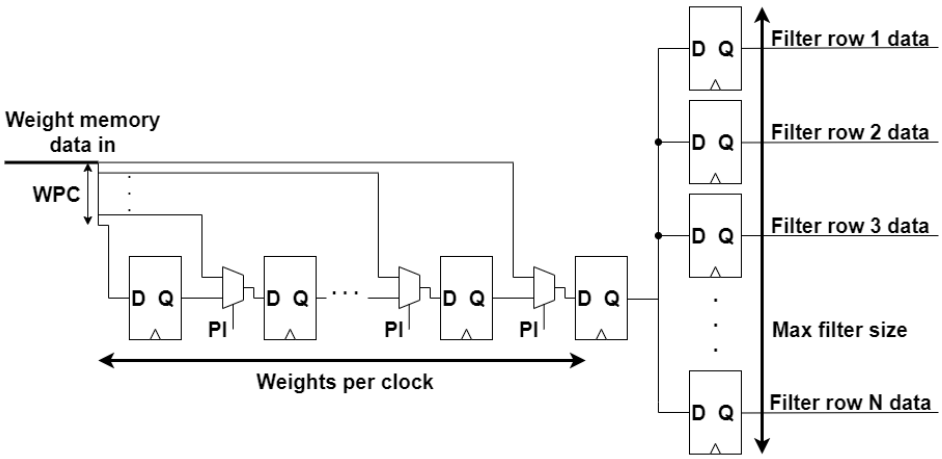

Figure 3.8: Weight structurer, alignes filter data form memory with buffers.

the computation block when a new computation cycle starts. This preemptive loading of the weights, instead of reading directly from memory, ensures to not bottleneck the architecture from the memory bandwidth. The buffers are each accommodated by a "filter structurer" block. The general block structure is seen in Figure 3.7. Internally, the filter buffers are a **MFS** by **MFS** grid of **DW**-sized registers connected to each other horizontally. This means that in order to load a new set of weights into the computation block, only **MFS** number of clock cycles are needed. The number of filter buffers and filter structurers are one each per filter slice used, thus $\mathbf{PF} * \mathbf{PD}$ number of both is needed.

The filter structurers are controlled by a block called "weight link", also seen in Figure 3.7. The role of the weight link is to coordinate the filter structurers with the weight memory. It gives the address to the weight memory and orders a filter structurer to latch the resulting memory output. Then the memory address jumps to either load the next address, the address that contains the next depth value for the current filter, or jumps to load a new filter depending on how the parallelization parameters are set. The general arrangement of the filter structurers is seen in Figure 3.8. Here, the input of the memory is split up into **weight memory DW**/**DW** number of arrays. This number is referred to as **weights per clock** (WPC). The split-up input is distributed to **WPC** number of registers through **WPC**-1 number of multiplexers controlled by the pick input (PI) signal. The weights latched into the registers are fed forward. When a weight is in the rightmost register in the chain, the corresponding filter row register latches the value. This continues for **WPC** number of clock cycles, then the filter structurer waits until another set of weight memory data is latched into the registers and continues to feed the filter row registers. Once the number of filter row registers corresponding to the current FS to be calculated have their values, the outputs are latched by the filter buffer block after the structurer and the next column of weight are loaded. This does serialize the parallel input of weights from the memory. However, as long as there is some parallelization, that is **PF** or **PD** are bigger than one, some filter structurer will be saving values from memory while another is moving values to the buffers. For example: If a parallelization gives four filter structurers and the **weight memory DW** is 32 and **DW** is eight. This gives **WPC** equal to four. Then reading from memory every clock cycle until all weights are loaded will be possible.

Weight Memory Management

To ensure the weight link fetches the correct weights from memory, the weights need to be saved in memory in a certain way. An example of how the weights are saved is provided in Table 3.4. With a weight memory with **weight memory DW** equal to 32 and **DW** equal to eight, 32/8 gives four weights stored per address row in memory. The example shows how N number of filters with a size of five by five by two are to be saved in memory. The weights are shown as F_{fdrc}, where f equals the filter index, d equals the depth index, r equals the row index and c equals the column index in the given filters. Filters weights are saved starting in the rightmost column and saved on a row-by-row basis. When one full two-dimensional slice has been saved and the depth, d, or the filter index, f, is incremented, the storing of

the values starts on a new address row.

Table 3.4: Weight memory space, shows the general structure of weight data in memory.

Memory address	Memory data			
FBA	F_{1115}	F_{1125}	F_{1135}	F_{1145}
FBA + 1	F_{1155}	F_{1114}	F_{1124}	F_{1134}
FBA + 2	F_{1144}	F_{1154}	F_{1113}	F_{1123}
FBA + 3	F_{1133}	F_{1143}	F_{1153}	F_{1112}
FBA + 4	F_{1122}	F_{1132}	F_{1142}	F_{1152}
FBA + 5	F_{1111}	F_{1121}	F_{1131}	F_{1141}
FBA + 6	F_{1151}	X	X	X
FBA + 7	F_{1215}	F_{1225}	F_{1235}	F_{1245}
FBA + 8	F_{1255}	F_{1214}	F_{1224}	F_{1234}
FBA + 9	F_{1244}	F_{1254}	F_{1213}	F_{1223}
FBA + 10	F_{1233}	F_{1243}	F_{1253}	F_{1212}
FBA + 11	F_{1222}	F_{1232}	F_{1242}	F_{1252}
FBA + 12	F_{1211}	F_{1221}	F_{1231}	F_{1241}
FBA + 13	F_{1251}	X	X	X
FBA + 14	F_{2115}	F_{2125}	F_{2135}	F_{2145}

3.2.8 Loading Inputs

Row Buffer

Input data is buffered on a row-by-row basis, using the row buffer block shown in Figure 3.9. The buffer consists of registers with the same number of bits as **DW**. The total number of registers is determined by **MIS** to allow a full row to be buffered. The **IS** run time parameter sets the size for different layers and is here used to insert multiplexers in the register chain, resulting in the buffer being able to change in length depending on the size of the current layer.

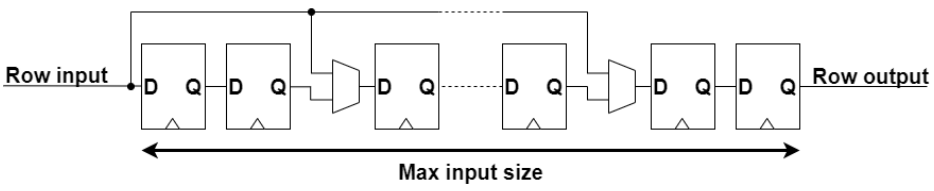

Figure 3.9: Row buffer, register chain of configurable length.

Row Buffer Connections

The output of the row buffers is connected to the row inputs of the computation block. The inputs and outputs of multiple row buffers are chained together, as seen in Figure 3.10.

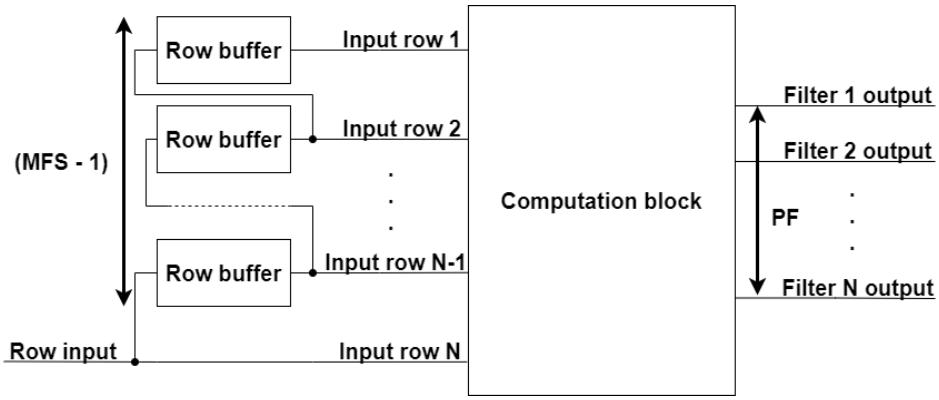

Figure 3.10: Row buffer connections to the computation block and to each other.

There will be **MFS**-1 row buffers created per **PD**. During an operation, the row buffers need to be filled with inputs, meaning the first row of the input data is read and fed to the row input and then the next row, etc. The input data will move up the chain of row buffers until **FS** (of the current layer) minus one number of row buffers are filled (since the bottom row is not buffered). This means that the first rows that the filter will move over are loaded, except the bottom row that goes directly from the input link output to the computation block without buffering. During computation, while the row data is outputted from the buffers to the computation block, the row data also move up to the next buffer. When the filter has moved over a complete row, and a new row below is loaded, no reloading of previous inputs from memory is needed. For every **PD** used in the design, loading and buffering the inputs is needed, thus the number of row buffers needed are $(\mathbf{MFS} - 1) * \mathbf{PD}$. Since the buffer size depends on **DW** and **MIS** the number of needed registers grows fast depending on all four mentioned parameters.

Input Link

Interfacing of the input memory is realized by the input link, as shown in Figure 3.11. The input link provides an address to input memory and receives the memory data. Similarly, to the weight link, the input memory data is split up over **input memory data width** / **DW** = **Inputs per clock** number of registers with one register chain per **PD**. Values are saved to the correct row of registers by switching multiplexers with the appropriate read depth (RD) signal. On the outputs of the weight link, registers are inserted to delay the output. Since one set of inputs from one row of one depth gets loaded per clock cycle, the delay registers make sure that when outputting the values, they arrive at the same time to the core input and

to the row buffers. Out depth 1 will have **PD** minus one chained registers after the output, Out depth 2 will have **PD**-2 chained registers after the output and so forth. In this input link, the maximum **PD** limit is seen, inputs per clock will be the maximum number of **PD** since if **PD** were to be any higher, the needed number of input values per clock exceeds the bandwidth of the input memory.

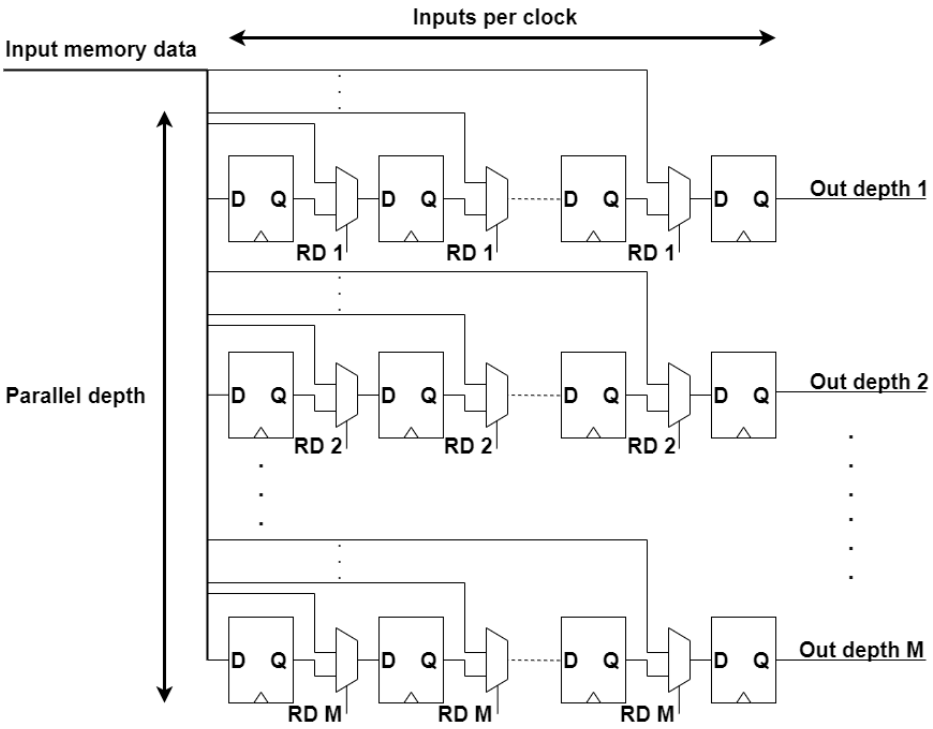

Figure 3.11: Input link, reads input memory data and outputs the data to the row buffers.

Input Memory Management

Structuring the input data in memory is performed in a similar fashion to weight management in the weight memory. An example is seen in Table 3.5. With **input memory DW** equal to 32, and **DW** equal to 8, giving four input values per address row in the memory. The indices in In_{drc} have d representing depth, r representing row, and c representing column of the input. An input with dimensions of three by three by some depth is given. When a new chunk of the depth is to be loaded, a new address row in memory is to be used (similar to how the weights were managed). One parameter not given in the run time parameters is Input Base Address (IBA). This parameter is not loaded pre-synthesis but given during the actual run time/operation of the unit to the top module in a "true run time" fashion, giving more flexibility as to where the inputs are stored.

Table 3.5: Input memory space, shows the general structure of input data in memory.

Memory address	Memory data			
IBA	In_{111}	In_{112}	In_{113}	In_{121}
IBA + 1	In_{122}	In_{123}	In_{131}	In_{132}
IBA + 2	In_{133}	X	X	X
IBA + 3	In_{211}	In_{212}	In_{213}	In_{221}
IBA + 4	In_{222}	In_{223}	In_{231}	In_{232}
IBA + 5	In_{233}	X	X	X
IBA + 6	In_{311}	In_{312}	In_{313}	In_{321}

3.2.9 Filter Result Accumulator

The results produced by the computation block are (depending on **PD** and input depth) a sum of some depth of the input per filter computed in parallel. For the times the input/filter is of a depth (both have the same depth) greater than the **PD** used, the result from the computation block is only a partial result. This partial result needs to be saved. The next part of the input depth needs to be calculated and added to the partial result. To store the partial result, two buffer memories per **PF** are used. Reading out the partial result and adding it together with the newest value from the computation block is performed by the filter result accumulator (FRA) block seen in Figure 3.12. For the first results of a filter, the buffer memory data inputs to the FRA are set to zero, since the first result does not need any previous result added. The result is stored in one of the two memories. After all partial results of the output are computed, the computation block starts to compute the next set of results for the same filters. Now the previously stored results will be read from one of the two memories where it was stored, added together with the new result from the computation block, and stored back to the other memory. This process continues until the full depth of an input/filter has been computed.

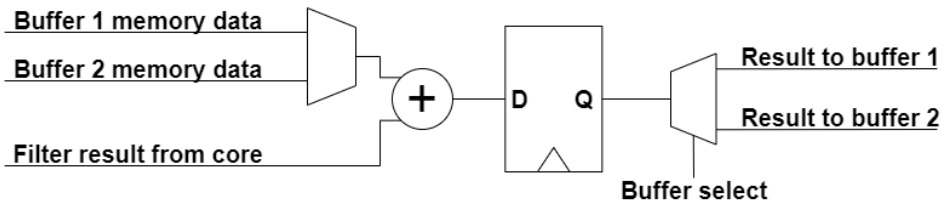

Figure 3.12: Filter result accumulator, selects a partial result from a buffer memory, sums it with the latest computed result and stores back to buffer memory.

Important to note is that one FRA accumulates results and stores them for one filter. This means that for every **PF** used one FRA block is added, and more importantly, two buffer memories are needed. The size of these memories needs

to be big enough to store all output results of one filter. For example, if the number of output results from one filter equals 400, the buffer memories need to be able to store at least 400 results each. To ensure that during accumulation no overflow is generated, the buffer memories store one result per address row. The number of bits needed for the result out from the computation block was previously $2 * \mathbf{DW} + 2 * \log_2(\mathbf{MFS}) + \log_2(\mathbf{PD})$. Now with arbitrary many accumulations depending on the depth of the input, it is up to the user to make sure that the **DW** of the buffer memories is adequate to store the full result. In order to facilitate this, the buffer memories and the bias memory need to have the same DW. Usually, bias values are full 32 bit or 64 bit precision and thus if these sizes are used, the accumulation in the FRA hopefully does not overflow. However, if the accumulated values have a chance of growing larger than $abs(2^{31})$ or $abs(2^{63})$ (one bit used for sign), buffers and bias memory with larger **DW**s are needed. It is up to the user to ensure that the buffers and bias memory are large enough/adequate.

3.2.10 Writeback Unit

After a filter has been fully computed, the results are either in buffer 1 or buffer 2 for each filter and needs to be written to memory. This is performed by the writeback unit seen in Figure 3.13. The writeback unit takes control from the FRA of the buffer (1 or 2) that holds the completed results. The computation block can start to compute the next set of filters and store the first partial results in the unused buffer, meaning that computation does not stop when saving the results. Inside the writeback unit, a few different processes take place. Firstly, the bias value for the current filters is added to the read results. The results then go through a truncation unit, going from a bit array length equal to that of the buffer memories/bias memory to a bit array length of **DW**. Relu is also performed in the truncation unit if needed. Secondly, the different filter results need to be delayed slightly since all results first arrive at the same time, but only one output address is written to at a time. To do this, results from filter two are delayed one clock cycle, filter three results are delayed two clock cycles, etc. After the delay registers are **output memory DW/DW = outputs per clock** number of registers that hold the results. After outputs per clock number of clock cycles the filter one row registers are filled, thus the multiplexers on the output select all values from the filter one row and outputs to memory. Because of the added delay, in the next clock cycle, filter row two has all the outputs per clock number of values. A jump is made in memory and filter two is saved. The first filter starts saving at **RSA**. The outputs are saved in memory similar to how the inputs were saved. The example provided for the input memory in Table 3.5 presents the structure of the outputs in memory. The only change to be made from that example is that the d in In_{drc} instead represents the filter number instead of the depth since the individual filter results do not have a depth. The writeback unit sets an upper limit to **PF** and that is outputs per clock. **PF** cannot be greater than outputs per clock since the number of results arriving then exceeds the output memory bandwidth and a "build up" of outputs occurs.

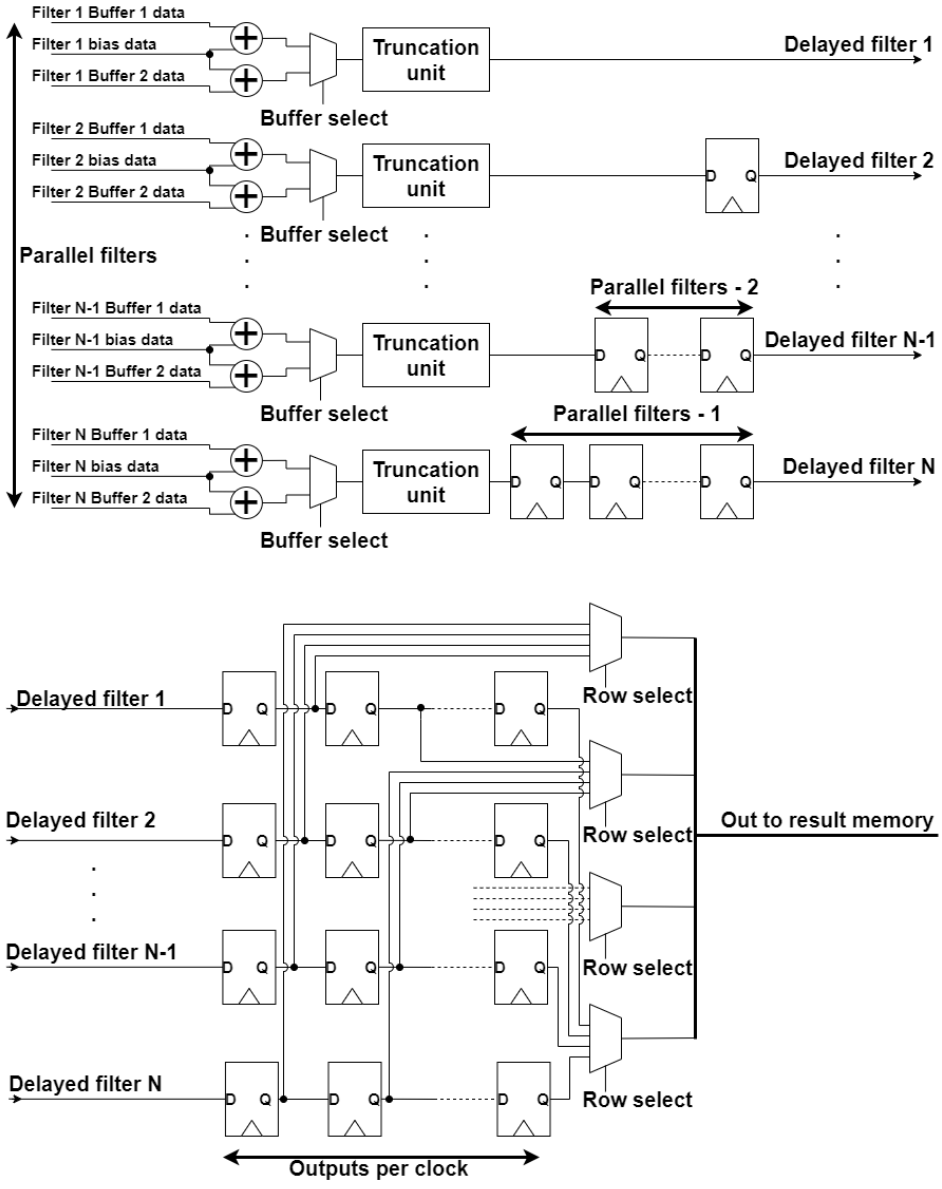

Figure 3.13: Writeback unit, reads results from buffer memory to then add bias, truncate, and save result to memory.

Truncation Unit

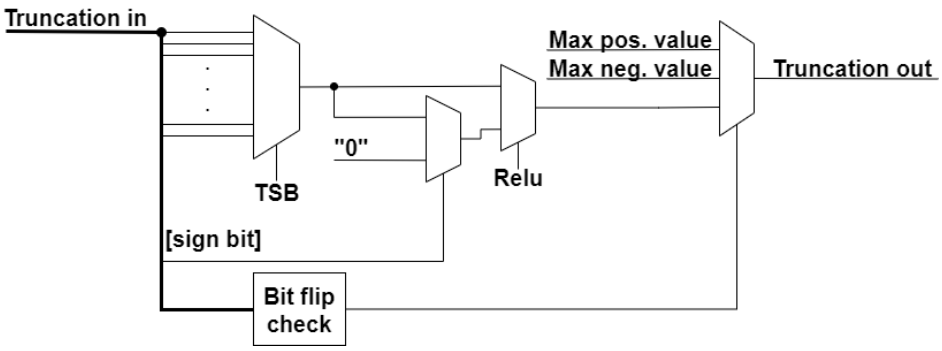

Figure 3.14: Truncation unit, truncates result at wanted bit position and applies activation function.

The previously mentioned truncation unit is seen in Figure 3.14. The input is split up into **DW** sized arrays. For example, if the input is 32 bits, and the **DW** is set to eight bits, the input is split into 24 arrays starting from bit 31 (sign bit is ignored) down to (and including) bit 24, 30 down to 23, 29 down to 22, etc. until the lowest range, eight down to one. Then one of these ranges is picked by the multiplexer specified by the user from the **TSB** parameter. The sign bit is used to produce the relu value (if negative, output zero). The user-specified parameter **relu** is used to either output the relu result or just the truncated result. Finally, since networks are trained on a set of input data, but the actual inference inputs are completely new inputs, there is a risk of an input producing a result greater than the truncation point. This results in outputs of a small positive or negative value when the real output was meant to be the largest possible output. Thus, some logic is added to check all bits above the chosen **TSB**. If any bit has flipped to a one for positive numbers or if any bit has flipped to a zero for negative numbers, the output is changed to be adequate to the range that is representable with the current **DW**.

Bias Link

The bias link is a unit that addresses the bias memory and provides the bias values to the writeback unit. At the start of each filter computation, the bias link reads **PF** number of bias values from memory and stores them in the same number of registers feeding the writeback unit. The bias values are to be stored in memory starting with the bias for filter one at **BBA**, the bias value for filter two at $\mathbf{BBA} + 1$, etc. All multiplications in the computation block are implemented as two's complement integer multiplication. If the user wants to represent the data as having a fixed radix point, consideration of the bias value data must be performed. If the inputs and weights have some parts of their **DW** size as integer representation and decimal representation (i.e., fixed point representation), the radix point will be shifted after multiplication and needs alignment with the bias

value radix point. This can be explained by an example: Using **DW** equal to 8, multiplication of the input and weight will result in 16 bits. If the input values have four bits to represent decimal values, and the weights have six bits to represent decimal values, there are now ten bits after the radix point to represent decimal values after the multiplication. The bias values in memory then need their radix point physically aligned with the radix point of the multiplication at bit ten, in order to be added correctly later in the writeback unit. An illustration of this is given in Figure 3.15.

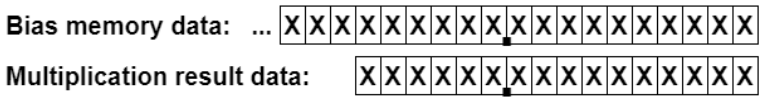

Figure 3.15: Bias and multiplication radix point line up for addition.

3.2.11 Overview of Top Module and Memory Connections

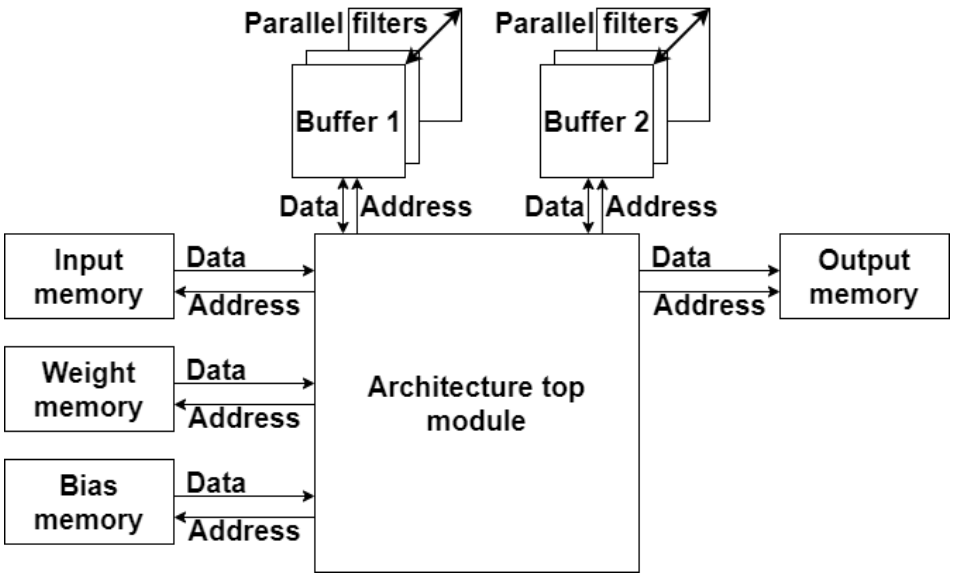

Figure 3.16: Architecture top module with memory connections.

The top block and its memory connections is shown in Figure 3.16, presenting an overview of the top-level connections of the system. If input and output memory are kept to the same **DW**, it is possible to add multiplexers and a control bit to the top-level design, allowing the user to switch input and output memory. This grants the ability to run consecutive convolutional layers without moving the results from the output memory to the input memory.

Chapter 4
Compiler Generated Example

This chapter details different implementation results from examples synthesized from architectures generated by the designed compiler.

4.1 Hardware Specification

Table 4.1: Specified hardware parameters supplied to the compiler.

Memory parameters	
Input AW, DW	13, 32
Weight AW, DW	13, 32
Bias AW, DW	13, 32
Buffer AW, DW	13, 32
Output AW, DW	13, 32

Accelerator parameters	
Data Width	8
Max. Filter Size	5
Max. Input Size	32
Max. Input Depth	32
Max. Number of Filters	32
Max. Stride	1

Values for the given HW parameters in the compiler are presented in Table 4.1. The same memory IP was used for all different memories; thus all width values are the same. Accelerator parameters were set accordingly as the maximums of the layers to be computed. The **PF** and **PD** parameters were changed to give multiple synthesized versions with different performances.

4.2 Runtime Specification

In Table 4.2, the runtime parameters of the convolutional layers used for the example are presented. One notable thing is the address used for **RSA**. The given memories have an address range of $2^{13} = 8192$ available for use. The convolution over layer one generates exactly 8192 results. This means that all available memory is used. Consequently, there is no need in starting to store results from subsequent layers at another point in memory than the first address. The assumption before starting a computation after layer one is that the previously generated results have been used or moved from memory. This problem does not exist for layer two and

Table 4.2: Specified runtime parameters supplied to the compiler.

Parameter name	Layer 1	Layer 2	Layer 3
Input Size	32	16	8
Input Depth	3	32	16
Filter Size	5	5	5
Stride	1	1	1
Padding	Yes	Yes	Yes
Number of Filters	32	16	32
Trunc. Start Bit	19	21	20
Relu	Yes	Yes	Yes
Filter Base Address	0	670	4256
Bias base Address	0	32	48
Result Save Address	0	0	1024

three since both results fit in the memory at once. Starting computation of layer three is therefore possible after the computation of layer two. This assumes all weights and inputs are pre-loaded for both layers. An alternative is to use a bigger memory to store the results.

4.3 Extracted Results

Different versions are synthesized with the HW specifications shown in Table 4.1, with **PF** and **PD** changed within their allowed ranges. A clock period of 5 ns was used.

4.3.1 Run Time Operation

The number of clock cycles required to compute each layer depending on **PF** and **PD** is presented with a table for each layer given in Table 4.3 through 4.5. It can be seen in the tables that the performance change is predictable with respect to changes in the performance parameters **PF** and **PD**. Since **MFS** has been set to five, the number of multipliers per **PD** or **PF** is 25. This gives $25 * \mathbf{PF} * \mathbf{PD}$ number of multipliers. This in turn, with **PF** and **PD** taking possible values between one and four, gives the possibility to change the number of instantiated multipliers in the design to between 25 and 400.

Looking at the table data one can see that an increase in **PF** has the same effect on performance. For example, double the **PF** (and thus double the number of multipliers) gives a performance that is close to twice as fast, in other words, the calculation takes half as many clock cycles to complete. The performance gain is not exactly linear with **PF**, increasing **PF** with a factor of two does not increase performance with exactly a factor of two, but is close to an increase with a factor of two. It is within one to four percent of the expected increase. Looking at the change in performance with respect to **PD**, the same does not hold. While in

some cases (especially going from one to two **PD**) double the **PD** means double the performance, but for many cases, the increase is between 60 and 90 percent of the expected increase. Since there is some overhead, both in loading input values and in computation, the change in performance is most likely linked to both the HW parameters and runtime parameters. Both the HW parameters and the runtime parameters will change the exact performance, and thus another layer or other HW setups might not see the same limit. One likely contributor to the performance limitation is an overhead of pre-loading input values to load the row buffers. This overhead does increase with **PD**, since there are more row values to pre-load. Referring to Figure 3.11, there are more rows of registers to load before values start to be given to the buffers. One outlier is the change in **PD** from three to four in layer one (Table 4.3). This is explained by the fact that the input to layer one only has a depth of three. This means that the extra **PD** does not offer a performance gain compared to **PD** equal to three.

Table 4.3: Computation time in clock cycles for layer one.

PF \ PD	1	2	3	4
1	240644	162436	83652	83716
2	121606	82310	42950	43398
3	84270	57276	30634	31096
4	61866	42442	23178	23626

Table 4.4: Computation time in clock cycles for layer two.

PF \ PD	1	2	3	4
1	377444	190276	131748	96580
2	189110	95430	66182	48806
3	141976	71728	50020	37072
4	94834	48106	33690	25114

Table 4.5: Computation time in clock cycles for layer three.

PF \ PD	1	2	3	4
1	132420	68356	52356	36100
2	66534	34310	26342	18630
3	45808	23676	18616	13468
4	33370	17482	13914	10474

4.3.2 Area

Contrary to the runtime measured in clock cycles, the area is technology dependent, and these are the results when synthesized to the CMOS 28 nm FD-SOI technology by STMicroelectronics [8]. The given area results in Table 4.6 give the area in relation to a base case, which is **PF** and **PD** equal to one. The given results only include the architecture that is generated by the compiler. Memories and buffer memories are not included in the given area since these change depending on the memory IP used. To give a point of comparison, the memories used for this example with their size of 8192 addresses of 32 bits per address have an area of 4,3 times that of the base case. Given that there are four memories needed for inputs and outputs, plus two times **PF** number of buffer memories, the total area of the architecture including memories is between 27.0 to 63.1 times the base case for (**PF**, **PD**) of (1,1) and (4,4) respectively. This shows that for this particular example memory size has the greatest influence on final area.

Table 4.6: Synthesis area in relation to the base case of PF, PD equal to one.

PF \ PD	1	2	3	4
1	1	1.877	2.752	3.617
2	1.645	3.101	4.558	6.004
3	2.291	4.330	6.374	8.399
4	2.931	5.554	8.174	11.280

4.3.3 Timing

Table 4.7: Critical path in relation to the base case of PF, PD equal to one.

PF \ PD	1	2	3	4
1	1	1.031	1.062	1.125
2	1.003	1.083	1.096	1.117
3	1.023	1.068	1.110	1.119
4	1.028	1.073	1.145	1.238

The critical path in relation to the base case of **PF, PD** equal to one, is presented in Table 4.7. The values generated here are also a result of using the CMOS 28nm SOI technology by STMicroelectronics. The minimum frequency for the synthesized gate-level netlist is above 350 MHz. The generated timing results show some change between different **PF** and **PD**. This change is explained by the fact that the critical path is within the control parts of the architecture and not

the data path. The reason that this is the case here is that the used multiplication precision (**DW**) was set to 8 bits. A higher precision might change the critical path to the point where the data path has the longest timing path. If this happens, no difference is seen in the critical path between different **PF, PD** settings.

4.3.4 Power Consumption

A simulation to estimate the power consumption was carried out with **PF** equal to one and **PD** equal to three. The estimated power consumption at a frequency of 200 MHz for the whole circuit was 29 mW, with memories consuming the majority of the total power. Putting together the number of operations in a layer, the number of cycles to compute a layer, frequency, and the power consumption, one gets just above 200 GOPS/W (counting one multiplication as one operation). These results are estimated from the synthesized design. Place and route is outside the scope of this thesis but if carried out, wires and parasitics could be accounted for in simulation further improving the power consumption estimation of the architecture.

Chapter 5
Conclusion

To conclude, a near-memory neural network accelerator compiler has been produced within the frames of this thesis. A compiler that takes a number of configurable parameters to allow variable performance has been produced. This thesis presents the process of how a flexible architecture is designed and implemented in VHDL for different properties of convolutional layer computations in CNNs, without the overhead required in the specialization for each use case.

5.1 Limitations and Future Work

There are endless ways to improve the architecture and features that could be added. Presented here are a few ideas for features and improvements that can be explored in future work.

Add support for pooling. This is a good expansion since pooling layers often follow the convolutional layers. The pooling can be seen as a reduction in the output size of the convolution, performing the pooling before storing back the results in memory allows for a greater increase in **PF** since less output memory bandwidth will be needed to store back the results.

Memory integration. Currently, the compiler gives a top-level VHDL file that the user must connect the memories to. Streamlining this process with a script helps users without VHDL experience to more easily use the compiler.

Compiler Asserts. Many different rules regarding what value different parameters take were given in the accelerator architecture chapter. Currently, there are no checks implemented in the compiler for these rules. Implementing assert statements to check for rule violations helps the user avoid possible compilation errors.

Run time parameters. As mentioned in the thesis, the run time parameters are hard coded and do not change after synthesis. To allow the running convolutional layers to change post-synthesis, the hard-coded parameters must be moved to registers, enabling change at run time.

Register emptying. When one row of input data has been computed, the registers in the row filters need to output their values before the next row starts to be computed. Currently, to output the values, takes the same number of

clock cycles to complete as the size of the current filter being calculated. To remove this delay, a separate set of registers could be added. The next row of results would be saved to these registers while the first values are being outputted.

Bias value alignment. The current way of aligning the bias value in memory depending on how many fractional bits are used could be simplified. The place of the radix point could be given to the compiler and the alignment could be performed automatically. This allows the user to just store the bias value in memory without shifting it.

Bias value in weight memory. The weight link could be reworked to handle the loading of bias values. This allows for the bias values to be stored in the weight memory, reducing the number of needed memories by one. Assuming all weights and bias values fit in one memory, this would help save both power and area.

References

[1] Hyeokjun, C. et al. (2017) “Near-Data Processing for Machine Learning.”. Presented at 33rd Int. Conf. on Massive Storage Sys. and Tech. (MSST 2017), Santa Clara, CA, USA. Available: https://storageconference.us/2017/Papers/DifferentiableMachineLearningModels.pdf

[2] H. E. Sumbul et al., "A 2.9–33.0 TOPS/W Reconfigurable 1-D/2-D Compute-Near-Memory Inference Accelerator in 10-nm FinFET CMOS," in IEEE Solid-State Circuits Letters, vol. 3, pp. 118-121, 2020. doi: 10.1109/LSSC.2020.3007185

[3] S. Sun et al., "A Study of the Memory Wall within the Jacobi Iteration Method," 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 964-969, doi: 10.1109/HPCC.2012.140

[4] K. -T. Tang et al., "Considerations of Integrating Computing-In-Memory and Processing-In-Sensor into Convolutional Neural Network Accelerators for Low-Power Edge Devices," 2019 Symposium on VLSI Technology, Kyoto, Japan, 2019, pp. T166-T167, doi: 10.23919/VLSIT.2019.8776560.

[5] G. Singh et al., "A Review of Near-Memory Computing Architectures: Opportunities and Challenges," 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, Czech Republic, 2018, pp. 608-617, doi: 10.1109/DSD.2018.00106

[6] "IEEE Standard for VHDL Language Reference Manual," in IEEE Std 1076-2019 , vol., no., pp.1-673, 23 Dec. 2019, doi: 10.1109/IEEESTD.2019.8938196

[7] “Cadence Design Systems.”, Available: https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html (accessed Nov. 16, 2022).

[8] “28nm FD-SOI technology catalog - stmicroelectronics.”, pp.5, Available: https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/technology_presentation/group0/35/54/24/df/5d/39/4f/39/BRFDSOI0616/files/BRFDSOI0616.pdf/_jcr_content/translations/en.BRFDSOI0616.pdf (accessed Dec. 5, 2022).

[9] "PrimeTime Technology by Synopsys", Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime/technology.html (accessed Dec. 6, 2022).

[10] J. Singh and R. Banerjee, "A Study on Single and Multi-layer Perceptron Neural Network," 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2019, pp. 35-40, doi: 10.1109/ICCMC.2019.8819775.

[11] D. Dai, "An Introduction of CNN: Models and Training on Neural Network Models," 2021 International Conference on Big Data, Artificial Intelligence and Risk Management (ICBAR), Shanghai, China, 2021, pp. 135-138, doi: 10.1109/ICBAR55169.2021.00037.

[12] L. Tóth, "Phone recognition with deep sparse rectifier neural networks," 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, 2013, pp. 6985-6989, doi: 10.1109/ICASSP.2013.6639016.

[13] E. Kalali and R. van Leuken, "A Power-Efficient Parameter Quantization Technique for CNN Accelerators," 2021 24th Euromicro Conference on Digital System Design (DSD), Palermo, Italy, 2021, pp. 18-23, doi: 10.1109/DSD53832.2021.00012.

[14] J. Choi et Al., "Parameterized Clipping Activation for Quantized Neural Networks", 2018, doi: 10.48550/ARXIV.1805.06085

Printed by Tryckeriet i E-huset, Lund 2023

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-911
http://www.eit.lth.se