Memory Efficient Hardware Accelerator for CNN Inference

SERGIO CASTILLO MOHEDANO
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Memory Efficient Hardware Accelerator for CNN Inference

Sergio Castillo Mohedano
se5616ca-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor:
Joachim Rodrigues
Masoud Nouripayam
Arturo Prieto

Examiner: Pietro Andreani

May 15, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Convolutional neural networks (CNNs) have gained popularity in recent years due to their ability to solve complex problems in areas such as image recognition, natural language processing, and speech recognition. However, the computational cost and memory requirements of CNNs are significant challenges for their widespread deployment, particularly in edge devices where power and area budgets are limited. To address these challenges, this thesis focuses on the design of a low-energy CNN inference accelerator using near-data processing (NDP), which is an approach to improve energy efficiency by bringing computation closer to data.

This thesis presents a design for a CNN inference accelerator that utilizes NDP to improve energy efficiency. The accelerator is designed to execute convolutional layers of the CNN with high throughput and low power consumption. It uses parallel processing and data reuse techniques to reduce the amount of data transferred between the memory and the accelerator. In addition, clock-gating is applied to reduce power consumption. At *200 MHz*, it achieves a performance of *2.42 GOPS* and an energy efficiency of *47.54 GOPS/W*.

The accelerator is synthesized and simulated at gate-level to calculate its performance and energy consumption, and it is evaluated using the *CIFAR-10* dataset.

Overall, this thesis contributes to the field of CNN accelerators by providing a low-energy and high-performance design that could be used in edge devices for real-time CNN inference applications.

The design can be further optimized and customized for specific use cases, and it provides a foundation for future research in the field of NDP and CNN accelerators.

Acknowledgements

I would like to start expressing my heartfelt gratitude to my supervisors, Masoud and Arturo, who have been supportive throughout my entire journey. Their guidance, encouragement, and belief in my abilities have been instrumental in shaping the outcome of this master's thesis. I extend my sincere appreciation to my teacher, Joachim, for his support and the opportunities he has provided me to grow both professionally and personally.

Along this arduous yet fulfilling path of completing my master's thesis and the entire master's program, I would also like to express my gratitude to the exceptional colleagues with whom I have collaborated, studied, and spent countless hours working on our projects. Together, we have overcome challenges, exchanged ideas, and celebrated each other's accomplishments, creating a supportive and stimulating environment that has fueled my growth.

My heartfelt thanks also go to Carla, Jonathan, Zsófia, Océane, Katje, and others who have been instrumental in my stay and adaptation to Sweden. Their friendship, kindness, and generosity have made a significant impact on my experience, providing me with a sense of belonging and making me feel at home in a foreign land. Without their unwavering support, encouragement, and shared experiences, this journey would not have been as enriching and rewarding.

Furthermore, I would like to express my heartfelt gratitude to my dear friends Isaac, Fabienne, and others who, despite the distance that separated us, have been unwavering sources of support throughout this journey.

Lastly, I am forever indebted to my parents and brothers, whose unconditional love, unwavering belief in my abilities, and endless sacrifices have been the bedrock of my success. Their constant encouragement, sacrifices, and belief in my dreams have given me the strength and determination to overcome challenges and pursue excellence. I am eternally grateful for their unwavering support, guidance, and the countless ways they have shaped my character and nurtured my intellectual curiosity.

Popular Science Summary

Deep Neural Networks (DNNs) have achieved remarkable breakthroughs, exemplifying the potential to revolutionize the world we live in. One such example is the remarkable performance of *AlphaGo*, a computer program developed by *DeepMind* and based on DNNs, which achieved a historic victory against the world champion in the ancient and complex board game of Go in 2016. This accomplishment, among others, underscores the potential of DNNs to transform various fields and bring new and innovative solutions to real-world problems.

Convolutional Neural Networks (CNNs), a subset of DNNs, are used in today's world in many applications such as image classification, speech recognition, and natural language processing. CNNs have achieved significant milestones in the past years, such as surpassing human-level accuracy in complex image classification benchmarks such as *ImageNet*.

The design of efficient CNN accelerators is crucial to meet the increasing demand for real-time and energy-efficient processing. In this thesis, the design of a CNN accelerator for inference is proposed, which takes advantage of the Row-Stationary dataflow. This dataflow technique improves performance by reusing data and distributing it to Processing Elements across a Network On Chip, bringing data closer to the computation units.

The design is validated using the CIFAR-10 benchmark, which is a popular dataset used to evaluate CNN performance. The proposed design uses fixed-point quantization, which reduces memory usage and computation complexity while maintaining acceptable accuracy levels.

Overall, this thesis presents a step forward towards accomplishing a reliable, efficient and flexible accelerator. The proposed design addresses the challenges associated with CNN acceleration, such as high computational cost and memory bottleneck, while improving energy efficiency.

List of Acronyms

AI: Artificial Intelligence
ALU: Arithmetic Logic Unit
ANN: Artificial Neural Network
ASIC: Application-specific Integrated Circuit
CNN: Convolutional Neural Network
CPU: Central Processing Unit
DNN: Deep Neural Network
FC: Fully-Connected
FPGA: Field Programmable Gate Array
GPU: Graphic Processing Unit
HDL: Hardware Description Language
IC: Integrated Circuit
IoT: Internet of Things
LFSR: Linear Feedback Shift Register
MAC: Multiply-and-Accumulate
MC: Multicast Controller
ML: Machine Learning
NDP: Near Data Processing
NN: Neural Network
NoC: Network on Chip
PE: Processing Element
PISO: Parallel-In Serial-Out
RAM: Random Access Memory
ReLU: Rectified Linear Unit
RF: Register File
RN: Round to Nearest
RS: Row-Stationary (dataflow)
RTL: Register Transfer Level
SGD: Stochastic Gradient Descent
SoC: System on Chip
SRAM: Static Random Access Memory
VHDL: VHSIC Hardware Description Language

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's most trending workloads such as high-resolution image processing, big data analytics, and Machine Learning (ML) are facing increasing computational and memory demands. These applications require high performance, real-time processing and low power consumption. However, the classic *von Neumann* architecture is unable to cope with these new demands. To address these challenges, new ways of moving and consuming data have been developed in recent years.

Among those computing and memory hungry applications that are proliferating nowadays, those related with Artificial Intelligence (AI) are between the most trending ones due to their ability to achieve great performance on different countless of backgrounds. As an example, very recently *OpenAI* released *ChatGPT*, a language model which is able to answer questions in a conversational manner, give you snippets of code, write essays about any topic in a structured way, or even write you a poem with assonance rhyme [1]. Also from *OpenAI*, another model that has gained a lot of popularity during the last year is *DALL·E*, which is shaking the way of living of artists or graphic designers, among others. This model receives as an input the description of an image by using descriptive prompts and outputs such an image with outstanding results [2].

AI is the field in which machines are able to mimic and copy the behavior of humans and interact with their environment as a person would do, or even outperforming them. These machines process data and act in an autonomous way, without the need of any human interaction and are usually constrained to a specific application, not being able to perform any task beyond the one it has been programmed to do, this is also referred as Narrow Artificial Intelligence.

Machine Learning (ML) is a subset of AI that focuses on how algorithms process incoming data to predict outputs, and relies on the analysis of this data to refine the algorithm and improve the model. This process is also known as training. ML can be further divided into two categories: unsupervised learning and supervised learning. The main difference between these two is that supervised learning requires input data and its expected outcome (label) during the training

phase of the algorithm, while unsupervised learning does not require a label.

Neural Networks (NNs) are a specific type of ML model that mimic the behavior of neurons in the human brain. They consist of a large number of simple processing nodes, called neurons, which are interconnected with one another. These neurons are organized into layers, and a NN can be composed of just a few layers or dozens of layers. When a NN has many layers, it is usually referred to as a Deep Neural Network (DNN). Data movement in these networks is feed-forward, meaning data moves in a single direction from the input layer to the output layer. DNNs have been used in applications such as speech recognition, image processing, and autonomous driving to improve performance [3].

A specific type of NN are Convolutional Neural Networks (CNNs). A CNN is able of extracting features within different regions of the input data by locally applying filters or kernels, hereby it is possible to detect and record spatial and temporal dependencies in an image [4]. CNNs have proven to be highly effective at image recognition and processing tasks [5].

To gauge the accuracy of image classification networks, various datasets are utilized. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is a yearly event that assesses the performance of image classification and object detection algorithms using a subset of the ImageNet database, which contains more than 14 million images organized into over 21,000 categories. In 2017, SENet network model achieved a top-5 error rate (reflects the model's ability to correctly predict the correct label among its top five most probable predictions) of 2.251%. To give some context, for the same ImageNet dataset, Russakovsky et al. in [6] calculate an optimistic top-5 error rate for humans at 2.4%.

However, running CNN *inference* (i.e., using a trained CNN model to make predictions on new data) on edge devices, such as smartphones, tablets, and Internet of Things (IoT) devices, presents several challenges and requires careful consideration of a number of factors [7].

One challenge is the limited computational resources and power constraints of edge devices [8]. CNN models can require significant computational resources and power to run, which can be a challenge for devices with limited processing power or battery life. This can make it difficult to deploy CNNs on edge devices, especially for resource-intensive tasks such as real-time object detection or image classification.

Another challenge is the need for low latency. Many applications that rely on

CNN inference, such as augmented reality or autonomous vehicles, require rapid processing and decision-making. This can be difficult to achieve on edge devices, as the model must be able to process data and make predictions quickly without access to the computing resources of a cloud or server [8].

To address these challenges and meet the needs of running CNN inference on edge devices, it is important to carefully consider the hardware and software requirements of the application, as well as the trade-offs between performance, accuracy, and energy efficiency. Techniques such as model compression, quantization, and pruning can also be used to reduce the size and computational demands of the model, while still maintaining acceptable performance [9].

Within this context, the well-known *Memory Wall* problem began taking shape several decades ago. Despite improvements in technology processes for microprocessors, progress in memory technology has not kept up. As a result, the transfer rate of memory has not improved as quickly, leading to a bottleneck in system performance [10].

As long as improvements in semiconductor fabrication continued to progress, the Dennard Scaling Law and Moore's Law appeared to be sustainable solutions for dealing with increasing challenges and reducing the impact of the Memory Wall problem. However, during the first decade of the 2000's both laws started to tear apart; the frequency at which processors were functioning could no longer be increased while keeping power consumption constant, and the performance per watt could not be improved at the same pace as in the earlier years.

As a consequence, other systems beyond Central Processing Units (CPUs) have been coexisting together with other more specialized hardware, like Graphic Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs) or, more recently, Application Specific Instruction-Set Processors (ASIPs). The latter has been gaining popularity for finding the perfect equilibrium between the high performance achievable by ASICs while being able to get the desired flexibility of CPUs [11]. GPUs are widely used for tasks like image processing, gaming, and mining due to their ability to parallelize computations and increase performance compared to CPUs, even though they can still be a solution for general-purpose applications. FPGAs allows a higher degree of flexibility than GPUs and CPUs, as they can be reprogrammed while achieving higher power performance than CPUs. ASICs are the best solution when it comes to accomplishing a specific requirement, whether it is about being energy efficient or achieving low latency; they are, on the contrary, costly and have low flexibility. Figure 1.1 exemplifies this pattern.

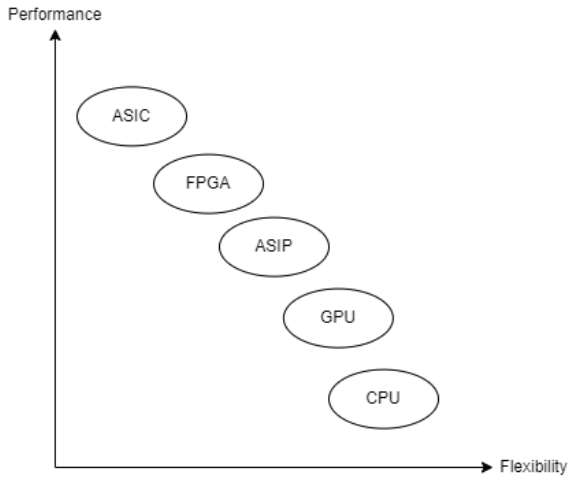


Figure 1.1: Flexibility vs. Performance of the main hardware architectures.

This paradigm has led to the development of alternative domain-specific architectures beyond traditional CPUs in order to address the Memory Wall problem and the challenges specific to running CNNs on edge devices. Therefore, designing an architecture that is both flexible and energy-efficient, while also exhibiting good performance, has consistently presented a challenge. The present work endeavors to address this challenge by creating a balanced and resource-efficient hardware architecture tailored for CNN inference.

1.1 Related Work

There have been a number of research efforts focused on the development of accelerators for CNN inference, with the goal of improving the performance and energy efficiency of CNNs.

One of the key contributions in this area was made by Chen et al. [12], who proposed *Eyeriss*, a hardware accelerator for CNN inference. *Eyeriss* is a domain-specific accelerator that is optimized for CNNs and uses a number of techniques to improve performance and energy efficiency, including data reuse, weight reuse, and fine-grained parallelism. Chen et al. demonstrated that *Eyeriss* was able to achieve significant performance improvements compared to traditional CPUs and GPUs, as well as reduced power consumption.

Eyeriss uses a Network-on-Chip (NoC), which is a core part that allows data delivery from the storage elements to the computation units by exploiting data reuse and reducing bandwidth requirements. The NoC allows communication between the Processing Elements (PEs) and the on-chip memory, as well as between the different processing elements themselves. The *Eyeriss* design introduced the use of a novel dataflow organization called *Row-Stationary* for the first time [13], which enables efficient data reuse and parallelism, being up to 2.5 times more energy-efficient than other existing dataflows. It should be noted, however, that some works such as [14] have argued that the selection of the dataflow architecture had a limited effect on the overall performance and energy efficiency of the accelerator, with other factors such as the design of the convolutional processing elements and the implementation of optimization techniques having a more significant impact.

The researchers at MIT who developed the *Eyeriss* accelerator also created an updated version, referred to as *Eyeriss v2*. This updated version presents several enhancements in terms of performance, energy efficiency, and chip area efficiency compared to the original *Eyeriss*. For example, when considering the AlexNet CNN model, and running at 200 MHz, *Eyeriss v2* demonstrates a higher throughput of up to 342.4 inferences/second, compared to the 34.7 inferences/second of the original *Eyeriss* [15].

Similar to *Eyeriss*, *ShiDianNao* [16] is a CNN accelerator which uses a scalable architecture that can support a wide range of neural network models and sizes. The architecture is based on a distributed computing model that uses a large number of PEs that can be interconnected to form a larger system. In addition, it uses a spatial architecture, where the PEs are arranged in a grid-like pattern. This architecture allows for efficient data sharing and communication between the PEs, which is critical for processing large neural networks.

Another notable contribution was made by Han et al. [17], who proposed the use of a hardware accelerator called *Deep Compression*. *Deep Compression* is a general-purpose accelerator that is designed to accelerate a wide range of CNNs and uses techniques such as weight pruning and quantization to improve performance and reduce the size of the model. Han et al. demonstrated that *Deep Compression* was able to achieve significant performance improvements and reduced model sizes compared to traditional architectures.

Overall, all these architectures revolve around the idea of Near-Data Processing (NDP). NDP is a computing paradigm that seeks to bring computation closer to the data. By reducing the distance that data has to travel, NDP can improve the performance and energy efficiency of the system by reducing the data transfer overhead between the accelerator and main memory. This can be achieved through the use of on-chip memory, such as scratchpad memory or register files, which can be accessed faster than off-chip memory. By storing the data and intermediate results of the CNN computation in on-chip memory, the accelerator can avoid the need to constantly access main memory, which can be a bottleneck for performance.

1.2 Thesis Outline

The goal of this work is designing a flexible and energy efficient hardware accelerator for running CNN inference.

The presented thesis is organized as follows:

- **Chapter 2: Background.** Introduces the reader to key concepts and specifications of CNNs, as well as describes different approaches towards accelerating the computation of such algorithms.
- **Chapter 3: Accelerator Design and Implementation.** Details the designed accelerator and its architecture.
- **Chapter 4: Baseline CNN Model.** Describes the process and approach followed regarding selecting a reliable CNN to validate the accelerator.
- **Chapter 5: Verification.** Presents the validations process for the functionality of the accelerator.
- **Chapter 6: Results.** Reports the results obtained in terms of throughput, power efficiency and area.
- **Chapter 7: Conclusions & Future Work.** Summarizes the general attributes of the accelerator, acknowledging potential areas for improvement, and outlines a number of enhancements that could be implemented.

Chapter 2
Background

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs), also referred as NNs, are a type of AI that are inspired by the structure and function of the human brain. The idea of NNs has its roots in the 1940s, when researchers first began investigating the possibility of using computers to model the behavior of neurons in the brain. One of the key early contributions to the development of NNs was the work of Warren McCulloch and Walter Pitts, who published a paper in 1943 outlining the concept of using artificial neurons to perform logical operations.

NNs are composed of interconnected artificial neurons that process and transmit information. In a NN, weights and biases are parameters that determine the behavior of the network. Weights are numerical values assigned to the connections between neurons in the network. Each weight represents the strength of the connection between two neurons and determines the influence that one neuron has on the output of another neuron. Biases are numerical values added to the inputs of each neuron in the network. The learning process of a CNN is simulated through the adjustment of weights and biases, allowing the network to adapt to new data and improve its performance.

The basic building block of the NN is the perceptron (or artificial neuron). Perceptrons are inspired by the biological neurons in the human brain, which receive input signals from other neurons and produce output signals. Figure 2.1 compares a biological neuron next to an artificial neuron. Input signals, coming from other neurons, are transmitted to a biological neuron through dendrites similar to the way an artificial neuron receives data from other perceptrons by connecting its inputs and outputs. Inputs in a perceptron are scaled through the use of weights, which reflect the significance of each input, this is analogous to the connections between the biological neuron and its inputs via synapses. The nucleus of a biological neuron generates an output signal based on the signals received through the dendrites, while the nucleus (node) in a perceptron performs calculations based on the input values and generates an output. Lastly, this output is carried away

by the axon in the biological neuron, and spread out to other neurons through the synapses.

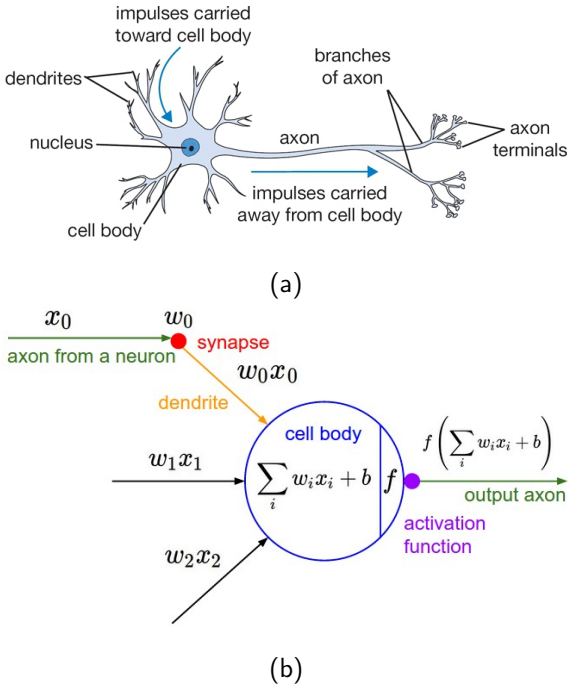


Figure 2.1: A cartoon drawing of a biological neuron (a) and its mathematical model (b) [18].

In the most basic form of a NN, perceptrons are disposed in layers, and they receive inputs from perceptrons belonging to the previous layer in a feed-forward manner. The inputs are then processed by multiplying them by their corresponding weight. A bias is then added to the weighted sum to fine-tune the result. Finally, non-linearity is added to the network using an activation function so that the model is able to adapt to complex problems. Another important characteristic of an activation function is that it controls the way a perceptron activates or fires. Equation 2.1 models this behaviour for a single perceptron computing the weighted sum of inputs $x_1 \dots x_k$, each of them multiplied by its corresponding weight $w_1 \dots w_k$, a bias b is added to the sum and an activation function σ is applied to the result, obtaining the output y.

$$y = \sigma \cdot (\sum_{n=1}^{n=k} w_n x_n) + b \tag{2.1}$$

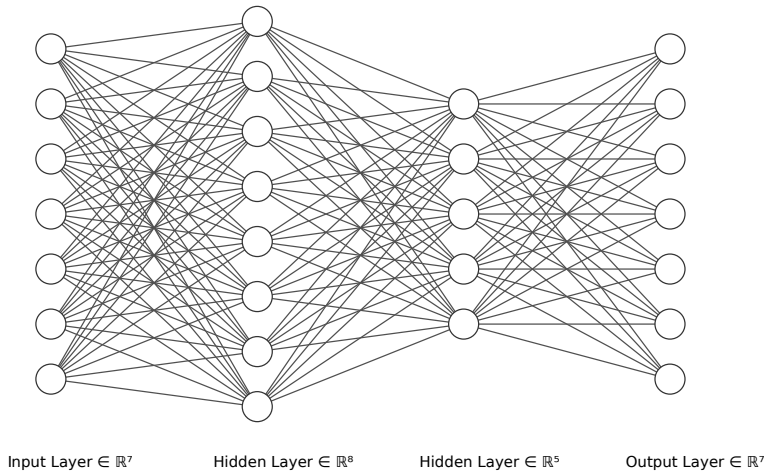


Figure 2.2: A Deep Neural Network with two hidden layers.

The output of the perceptron is then passed on to other neurons on the next layer in the NN. Figure 2.2 shows a simple model of such networks, where the first layer represents the input layer, the last layer is the output layer, with hidden layers in between.

Depending on the application, perceptrons may have different types of activation functions, such as tanh, sigmoid or Rectified Linear Unit (ReLU). ReLU activation function (eq. 2.2) is commonly used in deep learning. ReLU does not saturate for positive values, which means that it does not suffer from the vanishing gradient problem. The vanishing gradient problem occurs when the gradients of the weights in the network become very small, which can slow down or prevent the training process. The ReLU activation function avoids this problem because it does not saturate, so the gradients remain large and the network can continue to learn effectively.

$$f(x) = \begin{cases} 0 & x < 0, \\ x & x \geq 0. \end{cases} \tag{2.2}$$

Another important activation function which is worth mentioning apart is the softmax activation function (eq. 2.3), which is commonly used in multi-class classification problems such as those encountered in popular datasets used as benchmarks for CNNs. It converts the $j_1 \dots j_N$ raw outputs from a layer of a neural network into probabilities, with the sum of all values in that layer totaling 1. When softmax is used in the last layer in a NN, each individual output value represents the probability of the input belonging to a particular class. Another activation function is argmax, which also activates the perceptron in the output layer corresponding to the input's class, but it does so by one-hot encoding the output.

Hereby, unlike the softmax activation function, it does not have the capacity to evaluate the quality of the prediction.

$$softmax(x)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} \cdot e^{z_j}} \tag{2.3}$$

As an example, assuming there is a trained NN used to evaluate an input image of a dog, the output layer of the NN has three nodes with each one triggering depending on the possible outcomes or classes for which the network has been trained (cat, dog or frog). Table 2.1 compares the output of the NN with both softmax and argmax as activation function. Notice how the probabilities of the softmax output adds up to 1.

Table 2.1: softmax and argmax outputs of a NN having the image of a dog as input.

Class	**softmax**	**argmax**
Cat	0.253	0
Dog	0.736	1
Frog	0.011	0

2.1.1 Inference & Training

Inference refers to the process of using a trained model to make predictions or decisions. The weights and biases of the model are already tuned and the network is able to classify inputs into classes or categories. Inference involves feed-forwarding the values received in the input layer towards the output layer of the network.

Training involves using not only forward propagation as for inference but also backpropagation. Considering a supervised learning model, backpropagation is used to adjust the weights and biases of the network to decrease the error between the labeled output and the actual output. This is achieved by propagating the error through the network from the output layer, working backwards through the layers, and using the error to compute the gradient of the weights and biases, which are then updated to reduce the error by using optimization algorithms such as gradient descent. Together, these two methods are repeatedly employed to train the network until it reaches an acceptable level of accuracy.

2.2 Convolutional Neural Networks

The neuron (also known as perceptron), as the simplest form of ML algorithm, serves as the foundation for more complex NNs. When the neuron of a hidden layer receives all outputs from the previous layer as inputs, that layer is referred to as a fully-connected (FC) or dense layer. When only a subset of the neurons in one layer are connected to the neurons of the next layer, the latter is considered to be a sparsely connected layer. When this subset is local, meaning the neurons are close to each other, it is referred to as a receptive field. This locality is the primary attribute of a convolutional layer [19].

A CNN is composed of multiple layers, including at least one convolutional layer. The convolutional layer uses filters that convolve with the input image to extract features. It also uses pooling layers, which reduce the spatial dimensions of the feature maps. At the end of the network, the feature maps are then fed into fully connected layers that perform the final classification. CNNs are known for their ability to recognize patterns and features in images, which makes them well-suited for tasks such as image classification, object detection and image recognition.

One of the most widely-used and well-known CNNs is AlexNet. AlexNet was the first CNN to achieve breakthrough performance on the ILSVRC-2012 image classification dataset. Specifically, it outperformed the second-best model by a substantial margin, achieving a top-5 error rate of 15.3%, compared to 26.2% for the next-best model [20]. In addition to AlexNet, there are other well-known CNNs such as MobileNet [21], which is designed for efficient mobile vision applications. It was first introduced by *Google* researchers in 2017. The main goal of MobileNet is to reduce the computational complexity and memory requirements of traditional neural networks, while still maintaining high accuracy in image classification tasks.

2.2.1 Convolutional Layer

In a network with multiple convolutional layers, each layer extracts distinct features from the images, and as the layers become deeper, the complexity of these features increases. This increased complexity typically leads to a higher number of channels in the feature maps, making the convolutional layer a high-dimensional operation. For example, the first layer may focus on the detection of edges in the image, while a deeper hidden convolutional layer may focus on detecting bird's beaks, which ultimately would classify the input image as a bird.

A convolution is a mathematical operation that involves sliding a small matrix, called a kernel or filter, over a larger matrix, called the input feature map

(ifmap). The purpose of this operation is to extract certain features or patterns from the larger matrix. As the kernel slides over the larger matrix, it multiplies each element of the kernel with the corresponding element in the larger matrix, and then adds up all of the products. This result is then stored in a new matrix, called the output feature map (ofmap). The size of the ofmap depends on the size of the kernel and the stride of the sliding operation. The stride determines the distance by which the filter is moved in each iteration and controls the spatial dimensions of the output feature map, with larger strides resulting in smaller output feature maps, and smaller strides resulting in larger output feature maps. Typically, stride values of 1 or 2 are used.

The structure of a convolutional layer is shown in Figure 2.3; the ifmap is a three-dimensional matrix with height H, width W and with number of input channels C. Each ifmap channel convolves with a kernel of weights with dimensions height R and width S. For computing one 2-D ofmap, a 3-D kernel must convolve with a 3-D ifmap, and every resulting 2-D convolution corresponding to each ifmap channel is accumulated altogether to generate one 2-D ofmap channel. There are M ofmap channels, each with height E and width F. Hence, the set of kernels that comprises one convolutional layer is a 4-D matrix of size $[R\ x\ S\ x\ C\ x\ M]$. A one-dimensional vector of biases can be added to the results. In addition to this, a batch (N) of multiple input feature maps can be computed together. Table 2.2 summarizes the different shape parameters of a convolutional layer.[1]

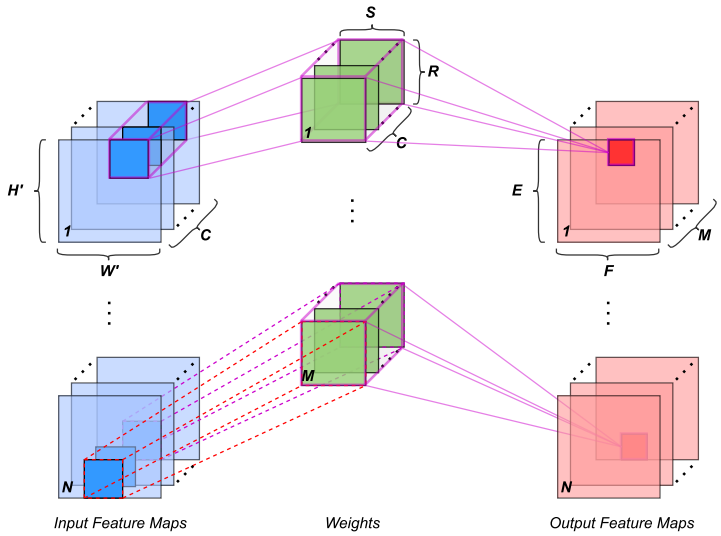


Figure 2.3: Structure of a convolutional layer [22].

[1]Notation of the different shapes and sizes, as well as naming conventions are based on the work of Sze et. al in [12, 19].

Table 2.2: Different shape parameters of a convolutional layer [22].

Shape Parameter	Description
N	Batch size of 3-D ifmaps
M	Number of ofmap channels (number of 3-D kernels)
C	Number of ifmap channels
H/W	Height/width of 2-D ifmap
H'/W'	Height/width of 2-D ifmap (including padding)
R/S	Height/width of 2-D weight
E/F	Height/width of 2-D ofmap

Padding is the technique of extending the dimensions of the ifmap channel by adding pixels (usually zeroes) to the border. This technique responds to two reasons. On the one hand, it helps preserving the information in the borders of the input image, as it allows multiple convolutions over these pixels. On the other hand, it ensures that ofmaps have the same spatial dimensions as ifmaps. This allows stacking multiple convolutional layers without reducing the spatial dimensions of the feature maps. If aiming to have same dimensions for ifmap and ofmap channels, it is important noticing that the padding to be applied to the ifmap will depend on the stride of the convolution.

2.2.2 Pooling Layer

The primary function of a pooling layer is to reduce the spatial dimensions of the feature maps, while also reducing the number of parameters and computational cost of the network.

The most common types of pooling layers are max-pooling and average pooling. From an area of the input feature map, max-pooling selects the maximum value, while average pooling computes the average value. These operations are applied to non-overlapping regions of the input feature map, resulting in a condensed output feature map.

The pooling operation is typically applied with a fixed window size and stride. The window size is the size of the region over which the pooling operation is applied, and the stride is the step size with which the window is moved across the feature map. These hyperparameters can be adjusted to achieve the desired reduction in the spatial dimensions of the output feature map. Figure 2.4 shows an example of the results of applying max-pooling and avg-pooling to a feature map of size $4x4$.

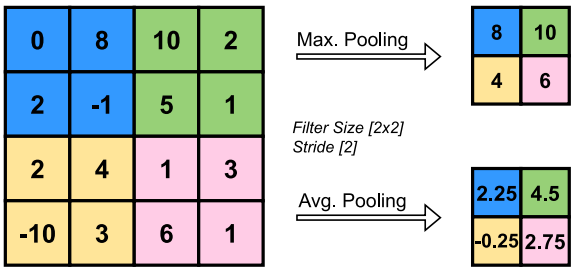


Figure 2.4: Avg-pooling and max-pooling operation with a stride of 2 and a size of $2x2$ over a feature map of size $4x4$.

2.2.3 Dense Layer

In a CNN, FC layers receive the output of the previous layers, which is typically a multi-dimensional array, and flatten it into a one-dimensional array, which can be used as input to the final output layer of the network. These layers are responsible for the final decision-making process of the network. A FC layer can be perceived as a convolutional layer which filters have the same dimensions as the input feature map ($HW = RS$), hence the output feature map's dimensions being reduced to one ($EF = 1$). With this constraints, it is evident there is no locality anymore.

2.3 CNN Accelerators

2.3.1 Spatial vs Temporal Architectures

As discussed in chapter 1, the myriad of applications proliferating within the last years related with ML shaped a context in which specialized architectures are being demanded to cope with the required needs. Consequently, for achieving the high parallelism needed in DNN algorithms, a distinction needs to be made between temporal architectures and spatial architectures.

Both architectures are composed of multiple processing units. However, control in temporal architectures is centralized, while the units of spatial architectures have their own separate internal control. Similarly, processing units in spatial architectures have an internal memory to store data, while processing units in temporal architectures have no memory capacity. Additionally, units in spatial architectures can be interconnected to each other to exchange data, allowing dataflow processing thus optimizing the reuse of data. To summarize, the computational units in temporal architectures are typically Arithmetic Logic Units (ALUs), while those in spatial architectures are more complex Processing Elements (PEs) that can support a greater variety of data movement patterns. Figure 2.5 shows a representation of both architectures.

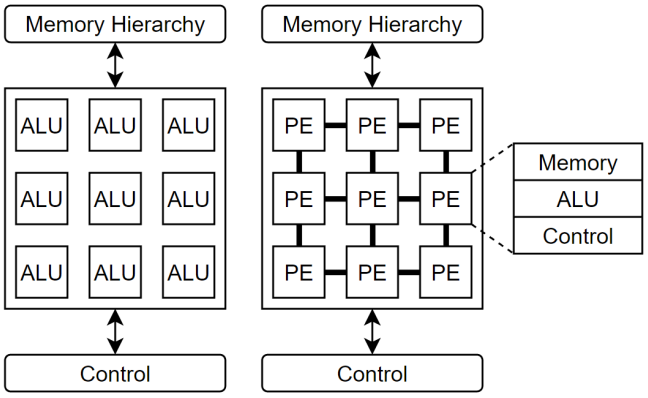


Figure 2.5: Temporal Architecture (left) and Spatial Architecture (right).

2.3.2 The Convolutional Operation as a Nested Loop

The fundamental computation for both convolutional and dense layers is the multiply-and-accumulate (MAC) operation. The commutative nature of this operation will allow a high degree of flexibility and parallelism in the way that the final ofmap pixels can be computed. The number of MACs that a convolutional layer has depends on the shape and size of the layer itself ($\#MACs = E \cdot F \cdot R \cdot S \cdot C \cdot M$). Similarly, $C \cdot M$ computes the number of MACs needed for a FC layer.

Algorithm 1 shows the convolution operation for a stride of 1, ignoring bias addition. It can be noticed that no matter the order in which the MACs are computed, the resulting ofmap value will remain unchanged. This phenomenon gives to the designer a high degree of freedom on how to achieve parallelism in order to compute as many MACs per cycle as possible.

Algorithm 1 Nested Loop of a Convolutional Layer

```
for m = 0; m < M; m++ do                          ▷ channel of ofmap
    for c = 0; c < C; c++ do                      ▷ channel of ifmap
        for e = 0; e < E; e++ do                  ▷ row of ofmap
            for f = 0; f < F; f++ do              ▷ column of ofmap
                for r = 0; r < R; r++ do          ▷ row of weight
                    for s = 0; s < S; s++ do      ▷ column of weight
                        ofmap[m][e][f] +=
                        ifmap[c][e + r][f + s] · weight[m][c][r][s]
                    end for
                end for
            end for
        end for
    end for
end for
```

2.3.3 Mapping and Design Space Exploration

Mapping is the allocation and scheduling of MAC operations onto hardware function units of an accelerator. Mapping is a threefold task. First, the architecture is defined with a computation dataflow according to the operations to perform. Then, a tiling strategy of each data type involved in the convolutional operation is tailored to the dataflow, which will impact the amount of data to be transferred to the PEs. Lastly, a binding strategy of the operations into the hardware is determined.

The evaluation of the trade-offs on the different characteristics and metrics of the accelerator is called Design Space Exploration (DSE). DSE is the process of identifying the optimal design for an accelerator that can efficiently accelerate the computations required for CNNs. This can include exploring different architectures, such as different types of memory hierarchies or interconnects, as well as different micro-architectural techniques, such as dataflow, tiling and unrolling parallelism.

A DSE strategy aims to find the best trade-off between performance, power consumption, and area-costs. This can be accomplished through the use of simulation and optimization techniques to evaluate the performance of different design options and identify the best one that meets the specific requirements.

The design space can be very large and complex due to the many different options available for the accelerator architecture and due to the nature of the NNs. Therefore, DSE requires a combination of automated exploration techniques and human expertise to guide the exploration and make informed design decisions [22, 23].

2.3.4 Row-Stationary Dataflow

A dataflow defines the way that data is moved through the accelerator and how it is processed by the different function units. It is utilized for exploiting the data reuse opportunities that are present in a spatial architecture. By using a multilevel storage hierarchy and maximizing the data reuse in the lower-energy cost levels, it is possible to reduce the data accesses of the higher-energy cost levels, such as the off-chip memory. More specifically, DNN algorithm allows data reuse opportunities, which include the utilization of filter weights across multiple input feature maps (filter reuse), the utilization of input feature map pixels across multiple filters (ifmap reuse), and the reuse of both input feature map pixels and filter weights as a result of the sliding-window processing in convolutions, known as convolutional reuse. Additionally, the intermediate values produced during the convolution operation, known as partial sums (psums), can also be reused by accumulating the output feature maps.

Nonetheless, taking into account that CNN accelerators are usually memory-constrained, it is not possible to implement both input data reuse (ifmaps and filters) and convolutional reuse simultaneously. For example, reusing input data for multiple MAC operations that compute different ofmap channels generates partial sums that cannot be accumulated together, thus requiring additional storage space. Consequently, the energy efficiency of the system is optimized when the mapping strategy effectively balances all types of data reuse within a multi-level storage hierarchy. Furthermore, there are different shapes and sizes of DNNs, and depending on the layer's size and type it might be more convenient to exploit some specific type of data over others. For example, dense layers would take advantage of input reuse rather than convolutional reuse [24].

Given the described scenario, Sze et. al [24] presented the *Row-Stationary* (RS) dataflow, which is designed to optimize all types of data movement for all data types across any level of the storage hierarchy in a spatial architecture, being up to 2.5 times more energy efficient than other dataflows such as weight-stationary or output-stationary.

The RS dataflow methodology segments the MAC operations into units referred to as mapping primitives. These primitives are executed on a single PE in a predetermined sequence and perform a 1-D row convolution (row primitive), maximizing data reuse for all data types. The 1-D row convolution consist on the MACs involved on one row of ifmaps with one row of weights. Figure 2.6 shows the computation of a 1-D row convolution within a PE. With this approach, it is possible to achieve several mapping choices that work both temporal-wise and spatial-wise.

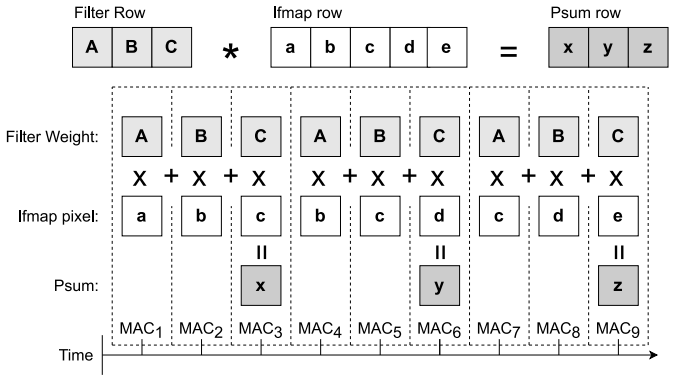


Figure 2.6: Row primitive running a 1-D row convolution within a PE [24].

One possible spatial mapping approach is to allocate the mapping primitives with data rows from the same 2-D plane within the PE array, which allows reusing the same rows of filter weights and ifmap pixels across PEs horizontally and diagonally respectively, and enables the accumulation of psum rows across PEs vertically. This mapping choice facilitates the execution of a 2-D convolution (as illustrated in Figure 2.7) and effectively exploits the opportunities for convolutional and psum reuse across primitives. Furthermore, if the PE Array is large enough, this mapping strategy can be duplicated within the PE Array, allowing the computation of several 2-D convolutions and exploiting further the data reuse opportunities.

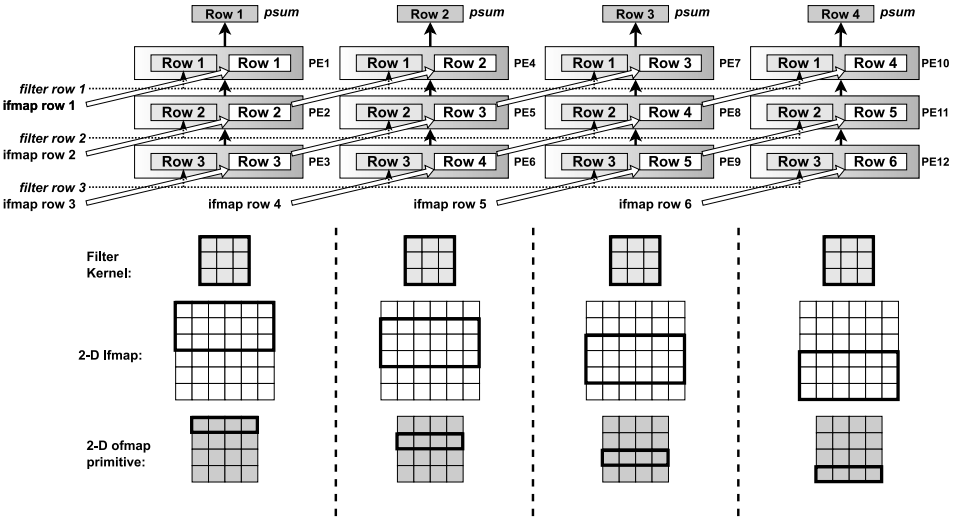


Figure 2.7: Patterns of how row primitives from the same 2D plane are mapped onto the PE array in the RS dataflow. Weight rows are reused vertically, ifmap rows are reused diagonally accross the PE Array, and psums are accumulated vertically [24].

The RS dataflow approach also allows for temporal mapping options such as the concatenation or interleaving of different row primitives within the same PE. For instance, it is possible to concatenate filter rows from different weight channels that convolute over the same row of ifmaps and generate partial sums belonging to different ofmaps. This increases ifmap reuse, but it also increases the on-chip memory required for storing intermediate partial sums.

When designing a CNN Accelerator with a spatial architecture, it is important to consider the flexible pattern of the convolutional operation and also the mapping opportunities that the RS dataflow offers. Parameters such as the number of PEs, on-chip memory size and shape, spatial and temporal mapping opportunities within each PE, should be taken into account.

2.3.5 HW Parameters of the CNN Accelerator

The group of PEs within the PE array that are needed for spatially mapping ifmaps, weights and psums for computing a 2-D convolution is called a PE Set. The dimension of a PE Set is exclusively a consequence of the shape and size of the convolution operation. For example, the PEs shown in Figure 2.7 conform a PE Set which size is given by the the height of the filter kernel ($R = 3$) and the height of the output feature map ($E = 4$). As stated in the previous section, if the size of the PE Array allows for multiple 2-D convolutions, it is possible to allocate several PE Sets within the PE Array, hence increasing spatial mapping capabilities.

Since the PE Set varies with the shape and size of the convolution operation, the accelerator shall be flexible enough to dynamically modify and allocate the resulting number of PE Sets on the PE Array to maximize data reuse.

To exemplify, let's take a PE Array which size is 3 x 32, and let's say we want to convolute an ifmap with size $H'W' = 10$ over a filter kernel which size is $RS = 3$, thus resulting in a primitive ofmap channel which size is $EF = 8$. Figure 2.8 shows that it is possible to fit 4 PE Sets, with each PE Set having a size of $R\ x\ E = 3\ x\ 8$, and each being able to compute a different 2-D convolution. Notice a primitive ofmap channel as the 2-D result of the convolution of a 2-D set of filters with an ifmap channel. The primitive ofmap channel needs further accumulation with other primitive ofmap channels before incurring in the final ofmap channel. Each PE Set computes a single primitive ofmap channel. By repeatedly computing different primitive ofmap channels the final multidimensional ofmap result is obtained.

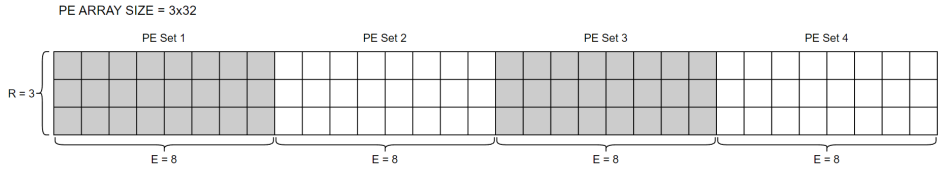


Figure 2.8: Allocation of PE Sets across a PE Array with size *3x32* for a convolution with a filter size of *3x3* and an ofmap of size *8x8*.

The chosen RS dataflow and the spatial architecture of the accelerator allows multiple spatial and temporal mapping options:

- *Unrolling* (r) the convolutional layer by computing different primitive ofmap channels from different 3-D weights belonging to the same ofmap channel, having the corresponding ifmap channel mapped in a different PE Set. This spatial mapping strategy increases the utilization of the PEs and also improves throughput of the computation path. Once the psums have been

computed on each PE Set, they can be further accumulated hence reducing data movement further.

- *Tiling* (t) the convolutional layer by casting the same ifmap channel to the PE Sets, and each PE Set having t different filters from different ofmap channels. Each PE Set computes different primitive ofmap channels from different 3-D weights belonging to different ofmap channels. Unlike with unrolling, there is no further accumulation of the psums computed by different PE Sets, since they all belong to different ofmap channels.
- Temporal mapping can be exploited by allocating more than one 1-D row of weights within the same PE. Consequently, a PE Set can process up to p 3-D filter weights over the same ifmap channel, with each 2-D convolution result corresponding to a different ofmap channel.
- Similarly, another temporal mapping option is to allocate several 1-D row of ifmaps within the same PE. In this case, each PE Set processes q different ifmap channels. This approach allows psum accumulation within the same PE Set, since it is possible to compute different primitive ofmap channels consecutively belonging to the same ofmap channel. However, internal memory within each PE for both ifmaps and weights would increase by q.

The parameters above are summarized in Table 2.3 below.

Table 2.3: Hardware Parameters of the accelerator [22].

Parameter	Description
p	Number of 3-D weights processed by a PE Set
q	Number of input channels processed by a PE Set
r	Number of PE Sets that process different input channels in the PE Array
t	Number of PE Sets that process different 3-D weights in the PE Array

A processing pass can be defined as the amount of computation needed for processing a single subset of the data that comprises the convolutional layer. The boundaries that determine the pass are based on that the input data (ifmaps and weights) can only be read once from the on-chip memory, and ofmap pixels are written back to the on-chip only once psum accumulation within the subset of computed data is finished.

The parameters that were just defined in Table 2.3 delimit the computational effort of each processing pass. Unrolling (r) and tiling (t) increases parallelism because there are more PE Sets available for processing, thus the total number of passes needed for computing a convolutional layer decreases.

On the contrary, with increasing values of the temporal mapping parameters p and q, in one pass the memory inside the PEs increase, as well as the number of 2-D ifmaps and weights processed (and so does the time). Reads and writes in between the on-chip memory and the PE Array also increase.

The accelerator's architecture is highly based on the concepts described in the present chapter. In chapter 3 a detailed description of such architecture is presented.

Chapter 3

Accelerator Design and Implementation

3.1 Overall Architecture

Figure 3.1 represents a block diagram depicting the high-level architecture of the accelerator. The core of the accelerator is the **PE**, which is organized in an array of **X** by **Y** PEs for increasing the parallelism. The **NoC** connects the memories with the PEs and uses the Multicast Controllers **MCs** to route the input data to the appropriate PE.

There are three separated SRAM-type memories for each of the three main data types involved in the convolution operation. The **Activation memory** allocates the ifmap pixels for one layer at a time, and it is updated by the ifmaps from next layer when convolution operation is finished for current layer. The **Weight memory** allocates all weights and biases for all layers, it also stores the configuration parameters from Table 3.8 per each layer. **OFMAP memory** stores all the ofmap pixels of a convolutional layer being computed until all accumulations have finished.

Once the final ofmap pixels are obtained (i.e. ifmap pixels from next layer), the bitwidth is lowered by applying *Round To Nearest* (RN) in the **ReLU/RN** block, in which also the ReLU activation function is applied. Lastly, **POOLING** is applied to the activation value, and the ofmap output, with or without max-pooling depending on the condition, is stored back to the Activation memory.

When the computation of one convolutional layer is finished, activations for next layer are already on the Activation memories, and the **SYSTEM CONTROLLER** points to the memory address in the Weight memory containing the weights, biases and configuration parameters of the next layer. The operation keeps going on until all convolutional layers have been computed.

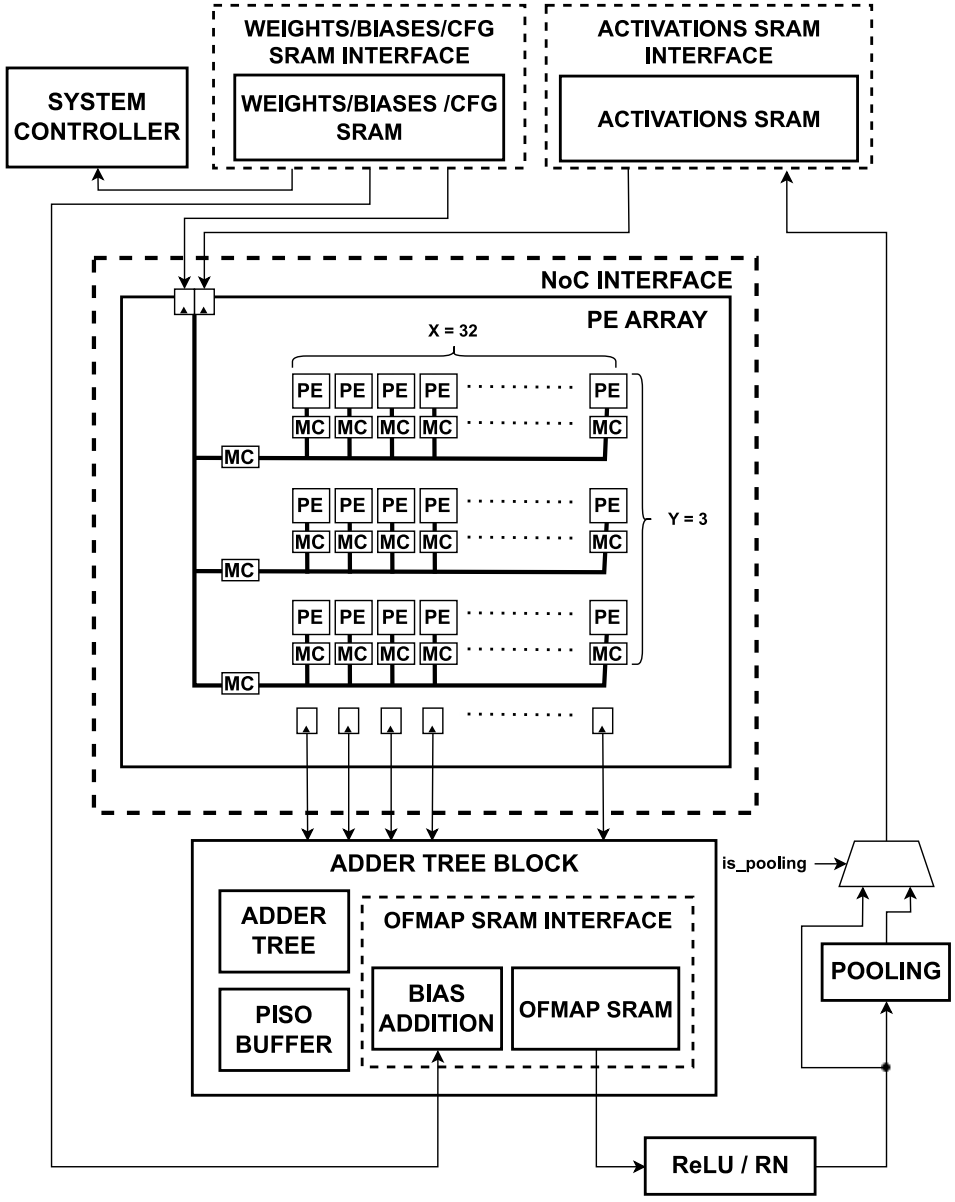


Figure 3.1: Block diagram of the accelerator, with a PE Array with size *3x32*.

3.2 Constraining the Mapping Space

Defining the mapping space of the accelerator requires some constraints which will limit the number of different convolutional layers that can be computed by the accelerator depending on their shapes and sizes. The goal is that, given some hardware constraints, the accelerator is adapted for those legal mapping points for which the accelerator is intended to work. The first criteria to consider is design time and scalability of the hardware rather than having a wide mapping space, and it is highly based on taking advantage of power of 2 divisions and multiplications, hence using shifting when appropriate.

The definitions of the parameters that define the dimensions of the convolutional layer and the convolution operation, as well as the parameters of the hardware accelerator, were introduced in Chapter 2, in Table 2.2 and Table 2.3.

Considering the accelerator's PE Array is composed of $X \cdot Y$ PEs, being X and Y the length and height of the PE Array respectively, then:

- *Same padding* is applied to the convolutional layers, meaning the height and length of the 2-D ifmap channel must be equal to the height and length of the 2-D ofmap channel:

  $HW = EF$

- The stride of the convolutional operation is limited to 1.

  $U = 1$

- Filter size must be an odd number lower or equal to the height of the PE Array[1]:

  $RS = 2 \cdot N + 1$, with
  $N \in \mathbb{N}$, and
  $RS \leq Y$

  As a result of the above-mentioned constraints, the padding can be computed as:

  $P = (RS - 1)/2$, hence
  $H'W' = HW + 2 \cdot P = EF + RS - 1$

- The maximum size of the 2-D ifmap/ofmap channel cannot be bigger than the length of the PE Array:

[1] Receptive Field's dimensions other than *3x3* are possible to implement in the accelerator as long as the constraints allow it. However, they have not been validated in the present work.

$EF \leq X$

This responds to the need of allocating at least a whole PE Set within the PE Array.

- The size of the 2-D ifmap/ofmap channel must be a power of 2:

  $\log_2(HW) = \log_2(EF) = N$, with $N \in \mathbb{N}$

- The maximum number of parallel PE Sets that can be allocated in the PE Array should be a power of two, and is defined to be 4. Although, future improvements in the accelerator's scalability could increase this number. In addition, the spatial mapping is only given by the unrolling parameter r, meaning that tiling (t) parallelism is not applied in the accelerator:

  $r \in \{1, 2, 4\}$
  $t = 1$

  For example, if $X = 32$, the possible set of values for EF (HW) are limited to the following set of numbers:

  $$\left.\begin{array}{l} r \in \{1,2,4\} \\ r = X/E \end{array}\right\} EF \in \{\frac{X}{1}, \frac{X}{2}, \frac{X}{4}\} \xrightarrow{X=32} EF \in \{32, 16, 8\}$$

  On the other hand, if $X = 64$:

  $$\left.\begin{array}{l} r \in \{1,2,4\} \\ r = X/E \end{array}\right\} EF \in \{\frac{X}{1}, \frac{X}{2}, \frac{X}{4}\} \xrightarrow{X=64} EF \in \{64, 32, 16\}$$

So far, the constraints mentioned only bound the mapping space in height (H/E) or length (W/F), but not in depth (C/M). Table 3.1 shows some of the possible valid mapping points given the already described constraints.

Table 3.1: Example of some valid mapping points given the first set of constraints.

X	**Y**	HW	EF	RS	r *(unrolling)*	t *(tiling)*
64	3	64	64	1, 3	1	1
64	3	32	32	1, 3	2	1
64	3	16	16	1, 3	4	1
32	3	32	32	1, 3	1	1
32	3	16	16	1, 3	2	1
32	3	8	8	1, 3	4	1
16	5	16	16	1, 3, 5	1	1
16	5	8	8	1, 3, 5	2	1
16	5	4	4	1, 3, 5	4	1

The following constraints limit further the mapping space. Specifically, they limit the dimensional depth of the ifmap and ofmap channels (C/M).

- The temporal mapping of rows from different ifmap channels within the same PE, given by parameter q, is set to 1, with no chance of taking advantage of this mapping strategy.

  $q = 1$

- The temporal mapping of rows from different weight filters belonging to different ofmap channels, given by parameter p, is variable and can be of up to 8[1]. The maximum value that parameter p can take is limited by the size of the Register File (RF) dedicated to the weights within the PE. In order to contain the size of the RF, this maximum the maximum value is chosen to be 8.

  $p = 8$

- The number of passes needed to process a convolutional layer must be an integer number, so as to avoid any idle PE during all processing passes. Otherwise, during a processing pass, there will be left an odd number of ofmap/ifmap channels to compute.

  The number of passes is given by:

  $$\#ofPasses = \lceil \frac{C}{r} \rceil \cdot \lceil \frac{M}{p} \rceil$$
  $$0 < r \leq C$$
  $$0 < p \leq M$$

Table 3.2 shows some of the possible valid mapping points considering the constraints just described.

Table 3.2: Example of some valid mapping points given the second set of constraints.

X	**Y**	HW	EF	RS	C	M	r	t	p	q	#Passes
32	3	32	32	3	3	16	1	1	8	1	6
32	3	16	16	3	16	16	2	1	8	1	16
32	3	8	8	3	32	32	4	1	8	1	32
16	5	8	8	5	128	64	2	1	4	1	1024

[1]Both the system controller and the local PE controller allows values other than 8 for the p temporal mapping parameter. However, they have not been validated in the present work.

Notice that the number of input and output channels, together with the temporal mapping (p) and the unrolling parameter (r) result in an even number of processing passes. Lastly, below there are few more constraints not directly related with the shape/size of the layers, but with the architectural characteristics of the accelerator, such as memory limitations or the activation functions implemented.

- The accelerator's hardware only allows max-pooling with a stride of 2 and a pooling window of size 2.
- The accelerator only supports ReLU as activation function.
- Depending on the size of the RFs in the PE, it is possible to vary the temporal mapping parameters described above. However, increasing these Register Files can worsen area-cost and energy-efficiency. Hence, as a design choice, the size of the Register File dedicated to the ifmap pixels is going to be determined by the CNN's convolutional layer that has the largest length (H). In the same way, the size of the Register File dedicated to the weight values is going to be determined by the width of the filter kernel (S) and the maximum allowable value for the temporal mapping parameter p; thus the size of this Register File is $S \cdot p$.

3.3 HW Parameters & Configuration Parameters

The baseline CNN Model that is used to evaluate the accelerator is described in chapter 4 and is summarized in Table 3.3. As it can be observed, all its convolutional layers fall within the valid mapping space of the accelerator described in previous section.

Table 3.3: Layer Parameters of the CNN Baseline Model.

Layer	C	M	HW	H'W'	RS	EF	U	P
Conv. 0	3	16	32	34	3	32	1	1
Conv. 1	16	16	32	34	3	32	1	1
max-pooling 0	16	16	32	32	-	16	2	0
Conv. 2	16	32	16	18	3	16	1	1
Conv. 3	32	32	16	18	3	16	1	1
max-pooling 1	32	32	16	16	-	8	2	0
Conv. 4	32	64	8	10	3	8	1	1
Conv. 5	64	64	8	10	3	8	1	1
max-pooling 2	64	64	8	8	-	4	2	0
Dense Layer	1024	10	-	-	-	-	1	0

The baseline CNN is composed of ten layers, out of which six are convolutional layers. Every two convolutional layers, max-pooling is applied to reduce the dimensionality of the feature maps. At the end of the network, the 3-D output of the *max-pooling 2* is flattened ($4 \cdot 4 \cdot 64 = 1024$), and all its values are fully connected to the ten neurons of the dense layers, which output's values are analyzed for classification purposes.

Table 3.4 below shows the arrangement of the accelerator based on the characteristics of each layer given that the dimensions of the PE Array are $X = 32$ and $Y = 3$. Table 3.5 shows the number of processing passes that each convolutional layer needs, and the number of MACs in total as well as the number of MACs to be computed on each processing pass.

Table 3.4: Accelerator Arrangement based on the characteristics of each layer, for $X = 32$ and $Y = 3$.

Layer	p	q	r *(X/E)*	t	# PE Sets *(t x r)*	PE Set Dimensions *(E x R)*
Conv. 0	8	1	1	1	1	(32 x 3)
Conv. 1	8	1	1	1	1	(32 x 3)
Conv. 2	8	1	2	1	2	(16 x 3)
Conv. 3	8	1	2	1	2	(16 x 3)
Conv. 4	8	1	4	1	4	(8 x 3)
Conv. 5	8	1	4	1	4	(8 x 3)

Table 3.5: # of MACs to be computed for each convolutional layer of the baseline CNN model.

Layer	**# Passes**	**# MACs** *(E x F x R x S x C x M)*	**# MACs/Pass** *(E x F x p x t) x (R x S x q x r)*
Conv. 0	6	442,368	73,728
Conv. 1	32	2,359,296	73,728
Conv. 2	32	1,179,648	36,864
Conv. 3	64	2,359,296	36,864
Conv. 4	64	1,179,648	18,432
Conv. 5	128	2,359,296	18,432

There are some design-time parameters to be set before passing the design through ASIC flow which are shown in Table 3.6. The table also shows the values given for evaluating the accelerator in the present work based on the baseline CNN Model, but they could be changed to encompass a wider mapping space and hence different CNNs.

Table 3.6: Hardware Parameters of the accelerator (I).

Parameter Name	**Value**
X	32
Y	3
hw_log2_r	(0, 1, 2)
hw_log2_EF	(5, 4, 3)
NUM_REGS_IFM_REG_FILE	34
NUM_REGS_W_REG_FILE	24
ADDR_4K_CFG	4042
NUM_OF_PARAMS	13

- **X** & **Y** are the length and height of the PE Array. For the evaluation of the accelerator in this work we set these values to 32 and 3, respectively, thus having a total of 96 PEs in the accelerator. As it has been discussed, the selection of these values allows us to map each of the convolutional layers of our Baseline CNN Model.
- **hw_log2_r** is utilized to multicast the ifmap pixels and weight values from their respective memories to the appropriate PEs through the *Multicast Controllers* (MCs). As it will be seen later in the chapter, the MCs route the input data dynamically based on the parameters of the layer being computed at that moment.
- **hw_log2_EF** is used to effectively divide the PE Array in r PE Sets on-the-fly depending on the parameters of the current convolutional layer

being computed. The accelerator takes all possible log_2 values of the different heights/widths for the valid mapping points (e.g. for our baseline CNN model, all possible values for EF are 32, 16 and 8). Both sets of hw_log2_r and **hw_log2_EF** values are hardwired in the MCs, and it would be possible to increase the mapping points by including a broader set of values without the need of modifying MCS's logic.

- **NUM_REGS_IFM_REG_FILE** & **NUM_REGS_W_REG_FILE** are the sizes of the RFs inside the PEs for the ifmap pixels and the weight values, respectively. The former is given by the layer which has the largest ifmap width, including the padding ($W' = 34$ for our baseline CNN model), the latter is set according to the temporal mapping parameter p and the width of the filter kernel S ($p \cdot S = 24$). Varying these parameters greatly affect the data storage elements contained within the PEs and hence the overall memory overhead of the accelerator, which ultimately also affects the area and energy efficiency, specially considering these memories are made of flip-flops. If a larger dimensions are to be computed for the accelerator, it might be needed to consider using other types of memories for the PEs, such as SRAMs.
- **ADDR_4K_CFG** points to the first memory address that contains the configuration parameters that are to be loaded during the operation of the accelerator. The different configuration parameters are to be loaded at the beginning of the layer computation.
- **NUM_OF_PARAMS** defines the number of configuration parameters per each layer in the CNN.

Table 3.7: Hardware Parameters of the accelerator (II).

Parameter Name	Value
ACT_BITWIDTH	16
WEIGHT_BITWIDTH	8
BIAS_BITWIDTH	16
PSUM_BITWIDTH	28
OFMAP_P_BITWIDTH	30
OFMAP_BITWIDTH	34
HYP_BITWIDTH	8

Not directly related with the arrangement of the accelerator, there are other hardware parameters to be considered which set the bitwidth of the input parameters (biases, weights and ifmap pixels) and the width of the data-path and the computation-path. These parameters are represented in Table 3.7.

- **ACT_BITWIDTH**, **BIAS_BITWIDTH** & **WEIGHT_BITWIDTH** are the bitwidth for activations, biases and weights, respectively. Using 16 bits for activation and bias values, and 8 bits for weight values responds

to the need of achieving an adequate accuracy similar to the one obtained during the evaluation of the CNN using floating values. Briefly, as elaborated in Chapter 4, shorter bitwidths degrades accuracy during fixed-point inference as compared to floating point inference.

- **PSUM_BITWIDTH** is set to avoid overflow of the *partial-sums* (psum) computed inside the PE Array.
- **OFMAP_P_BITWIDTH** refers to the bitwidth of the *ofmap primitive*, which is the ofmap value coming out of the PE Array block into the Adder Tree, where it is further accumulated with their corresponding ofmap primitives on their way to the PISO Buffer.
- **OFMAP_BITWIDTH** determines bitwidth of the ofmap, once all ofmap primitives have been accumulated.

In addition to design-time parameters which are set prior to hardware synthesis, there are also a set of configuration parameters that are loaded right before starting computation of the CNN and which are fetched from memory to the control logic for the accelerator to be configured and dynamically set its properties according to the needs of the convolutional layer being computed. Table 3.8 shows these parameters.

Table 3.8: Configuration Parameters of the accelerator.

Param. Name	Description	Value
L	*Number of (convolutional) layers*	
M	*Number of ofmap channels*	
C	*Number of ifmap channels*	
HW	*Height/width of ifmap channel*	
HW_p	*Height/width of ifmap channel, with padding*	
RS	*Height/width of filter kernel*	
EF	*Height/width of ofmap channel*	*as per tables*
r	*Unrolling factor (# of PE Sets*	*3.3 and 3.4*
p	*Number of different 1-D row filters to fit within a PE*	
M_div_pt	*Number of Passes until ifmap reuse*	
EF_log2	$log_2(EF)$	
r_log2	$log_2(r)$	
is_pooling	*1 (0) if there is (is not) max-pooling*	

L refers to the number of convolutional layers to be computed by the accelerator. The current version of the accelerator requires all the weights and biases from all layers to be loaded beforehand, and once computation starts it does so starting with the first convolutional layer. Right after the first convolutional layer has finished, ***is_pooling*** evaluates if there is (1) or there is not (0) a max-pooling layer after. If there is, max-pooling is applied to the activation outputs of the current convolutional layer being processed, then the next convolutional layer starts being computed. This process goes on continuously without the need of any external interaction until ***L*** convolutional layers have been computed. Each layer requires ***NUM_OF_PARAMS - 1*** for it to be computed by the accelerator. The -1

is because $\boldsymbol{L}$ needs to be loaded only once.

In the following sections a detailed description of the accelerator's hardware and its blocks is depicted.

3.4 System Controller

The System Controller's main function is setting up the nested loops of the convolutional operation according to the configuration parameters of the current layer being processed. The configuration parameters tailor the nested loop boundaries and its parallelization settings according to the needs of the layer as to maximize PE Utilization in the PE Array. Algorithm 2 shows the nested loop, similar to the one described in algorithm 1, but personalized to these parallelization techniques according to the mapping strategy used for the accelerator.

Algorithm 2 Nested Loops of Main System Controller

```
for c = 0; c < C − r; c+ = r do
    for m = 0; m < M − p; m+ = p) do
        for rc = c; c < c + r; rc + + do
            ********************                          ▷ IFMAPS NESTED LOOP
            for w′ = 0; w′ < H′W′; w′ + + do
                for h′ = 0; h′ < H′W′; h′ + + do
                    ifmap index [rc][h′][w′]
                end for
            end for
            ********************
            ********************                          ▷ WEIGHTS NESTED LOOP
            for r′ = 0; r′ < RS; r′ + + do
                for pm = m; pm < m + p; pm + + do
                    for s = 0; s < RS; s + + do
                        weight index [rc][r′][pm][s]
                    end for
                end for
            end for
            ********************
            ********************                          ▷ OFMAP NESTED LOOP
            for pm = m; pm < m + p; pm + + do
                for f = 0; f < EF; f + + do
                    for e = 0; e < EF; e + + do
                        ofmap index [pm][f][e]
                    end for
                end for
            end for
            ********************
        end for
    end for
end for
```

As it can be observed, there are three main nested loops that will trigger at specific times during the computation of the convolutional layer. At first, the RF's PEs are empty, the ifmap pixels and weight values corresponding to the layer being computed for the first processing pass are still in the memories, thus both the *IFMAPS NESTED LOOP* and the *WEIGHTS NESTED LOOP* start running concurrently.

The indices of the ifmap pixels start increasing in the *IFMAPS NESTED LOOP*, these indices are sent to both the Activation memory and to the MCs. The Front-End Read Interface of the Activation memory evaluates these indices and reads the appropriate activation value to be sent to the PE Array. On the other hand, the MCs receive both the ifmap pixel and the indices on the same clock cycle and evaluate to which PEs the ifmap pixel must be sent, so that a single read from the Activation memory is enough to send a specific ifmap pixel to the corresponding PEs according to the mapping strategy given by the Row-Stationary dataflow approach.

Similarly, the indices of the weight values also increase according to the *WEIGHTS NESTED LOOP*, these indices are sent to the Front-End Read interface of the Weight memory and to the MCs to multicast the weight values to the corresponding PEs.

There are three things to notice at this point. First, these two actions take place concurrently, meaning that the time it takes for the accelerator to load the input values to the PE Array is given by the longest Nested Loop, which at most cases is the *IFMAPS NESTED LOOP*. Second, there are processing passes in which there is no need to load the ifmap values, and the accelerator can reuse them for as long as the weight values can still be used. In these cases, only the *WEIGHTS NESTED LOOP* is run, while the *IFMAPS NESTED LOOP* remains inactive. Lastly, the *OFMAPS NESTED LOOP* triggers once the inputs values for the current processing pass are in the PEs, and controls when the processing pass is finished.

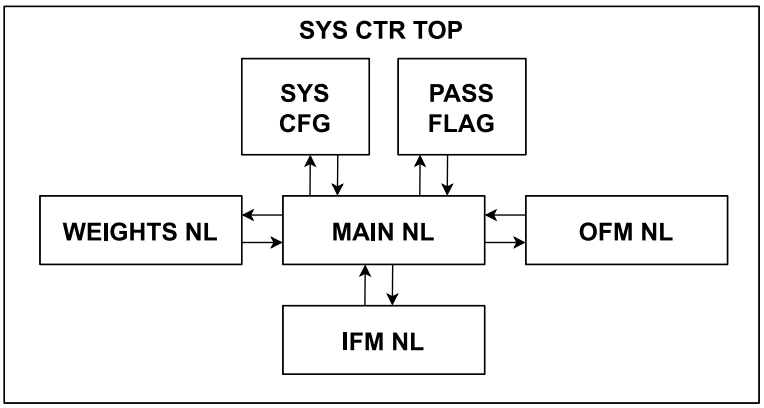


Figure 3.2: Block diagram of the System Controller.

Figure 3.2 represents a high level diagram of the System Controller and its different blocks. **SYS CFG** retrieves from memory the configuration parameters for the layer to be computed and stores them in registers for the whole computation of the layer. This parameters are used to set the limits of the nested loop's counters and configure different settings throughout the accelerator. **PASS FLAG** outputs a control flag that indicates when to trigger the computation and hence the *OFMAP NESTED LOOP*. **WEIGHTS NL**, **IFM NL** and **OFM NL** control the different nested loops already mentioned, and **MAIN NL** controls the outer parameters of the nested loops (c, m and rc) as well as manages the interaction between the different control blocks and the rest of the accelerator.

3.5 Memories

There are three memories that the accelerator uses for allocating the different data. The memories used in the present work are high-performance/high-density 28nm FD-SOI embedded memories provided by *STM*. More specifically:

- 1x Single-Port High-Density memory with 4096 words and 32 bitwidth.
- 3x Single-Port High-Density memory with 8192 words and 32 bitwidth.
- 8x Double-Port High-Density memory with 2048 words and 32 bitwidth.

The arrangement of the memories and total size of each memory block is shown in Table 3.9. The details of each memory block and the mapping of the data in the memories is described in subsections below.

Table 3.9: Arrangement of memories used by each memory block of the accelerator.

Memory Block	Memories used	Total Size [KB]
WEIGHTS SRAM	2x SPHD 8192x32 1x SPHD 4096x32	80
ACTIVATIONS SRAM	1x SPHD 8192x32	32
OFMAPS SRAM	8x DPHD 2048x32	64
TOTAL		176

3.5.1 Weight memory

The Weight memory block not only stores the weights of the network but also the biases and the configuration parameters of all layers in the network. The size requirements were given by the number of parameters of the Baseline CNN Model. The Weight memory allocates the parameters for the whole convolutional network, hence the user must ensure that the network size does not exceed the size of the memory.

The order at which the weights are stored in the memory is based on the mapping strategy of the accelerator; it depends on the spatial mapping parameter p and also on the RS dataflow.

Weight memory also stores the biases and the configuration parameters of the CNN. The hardware parameter **ADDR_4K_CFG** sets the memory pointer from which the configuration parameters are stored in memory, in increasing order of addresses. In a similar way, the biases are stored in decreasing order of addresses starting in the memory address **ADDR_4K_CFG - 1**. Figure 3.3 show a general picture of the memory mapping.

As Table 3.9 indicates, the Weight memory block is composed of three memories. The weights occupy the most of the memory block, leaving biases and configuration to be stored at the end of the memory block.

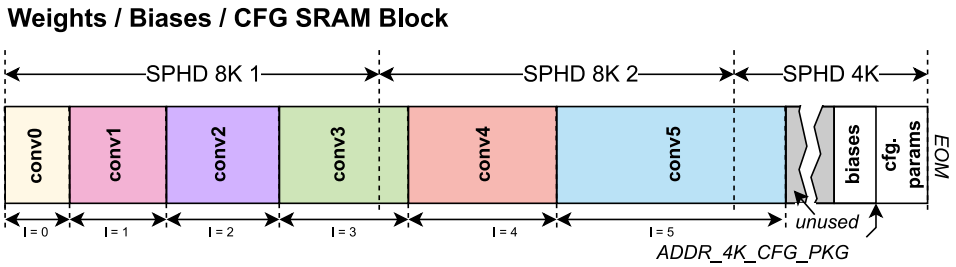


Figure 3.3: General View of the Weight memory Block.

The overall architecture of the Weight memory is shown in Figure 3.4. The Weight memory has a Front-End Read interface that handles the control signals as well as the indices coming from the system controller and the nested loops, and it retrieves the weights, biases and configuration parameters from the Back-End Interface, which directly interacts with the Wrapper Block and manages the addresses belonging to each of the three memories conforming the memory block. Notice there is not a Front-End Write interface since for the validation of the current version of the accelerator the parameters are forced in the SRAMs during simulation.

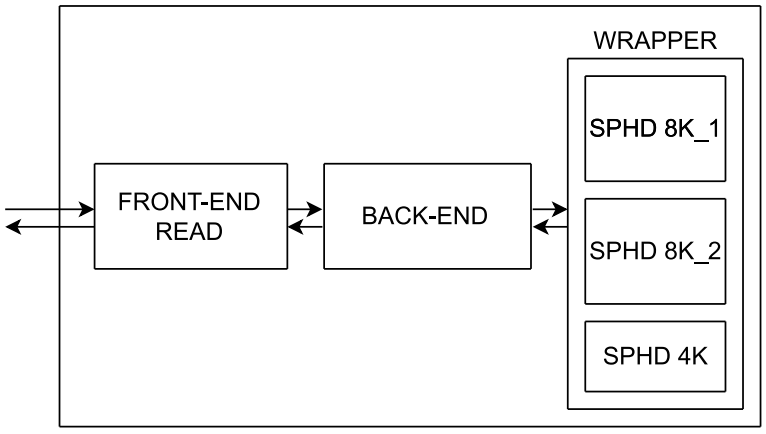


Figure 3.4: High Level Block Diagram of Weight memory.

3.5.2 Activation memory

The Activation memory allocates the ifmap values of each layer. At first, it allocates the ifmap pixels of the first image to run the inference with. When computation of the first convolutional layer finishes, the activations for the second convolutional layer overwrites the ifmap pixels, and this process is repeated until

all convolutional layers of the CNN are computed.

When there is a pooling layer followed by a convolutional layer, rather than storing the output activations of the convolutional layer, the activations stored are the ones computed after the pooling layer. As opposed to storing the activations directly after the computation of the convolutional layer and applying pooling afterwards, this approach allows the accelerator to prevent unnecessary reads/writes to the Activation memory. Given the Table 3.10, which shows the activations generated after each layer in the Baseline CNN Model, the activations in cursive are not stored in memory, but the ones generated after max-pooling is applied.

The size of the Activation memory was set according to the requirements of our CNN Baseline model. Being Conv. 0 layer the one generating the largest number of activations, and given that the activations bitwidth is 16, the memory needs at least 8,192 addresses to allocate all values.

Table 3.10: Activations generated per each layer of the Baseline CNN Model.

Layer	**Activations Shape** *M x E x F*	**# of Activations**
Input	[3 x 32 x 32]	3,072
Conv. 0	[16 x 32 x 32]	16,384
Conv. 1	[16 x 32 x 32]	*16,384*
max-pooling 0	[16 x 16 x 16]	4,096
Conv. 2	[32 x 16 x 16]	8,192
Conv. 3	[32 x 16 x 16]	*8,192*
max-pooling 1	[32 x 8 x 8]	2,048
Conv. 4	[64 x 8 x 8]	4,096
Conv. 5	[64 x 8 x 8]	*4,096*
max-pooling 2	[64 x 4 x 4]	1,024

The high level block diagram of the Activation memory is depicted in Figure 3.5. As opposed to the Weight memory, the Activation memory does have a Front-End Write interface, since the activations after finishing one convolutional layer must be stored back to the Activation memory for next layer's computation.

The padding that needs to be applied is not stored in the memories, since this would imply storing just zeroes unnecessarily. Instead, as shown in Figure 3.6 the indices h' and w' from the *ACTIVATIONS NESTED LOOP* are evaluated together with its control signals, and the outcome of the assessment serves as a control signal for the Back-End interface, which will read the appropriate value

from the memory, or will just send back a zero instead when padding. As it can be observed, the padding to be applied is dependent on the width/height of the filter kernel (RS) and the stride (U). The current version of the accelerator only works with strides of 1 for convolutional layers.

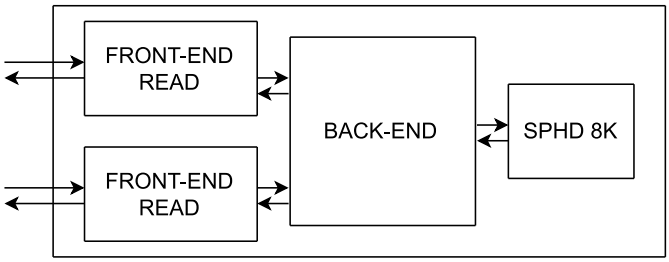


Figure 3.5: High Level Block Diagram of Activation memory.

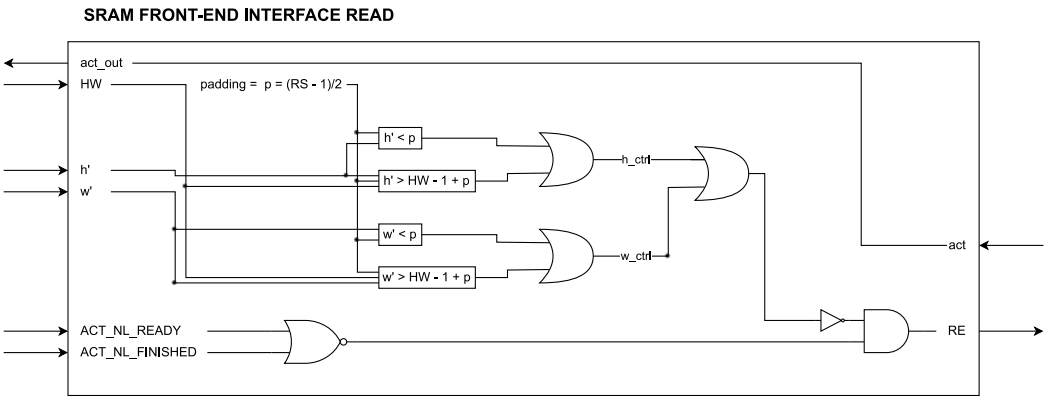


Figure 3.6: High Level Block Diagram of Activation memory's Front-End Read Interface.

3.5.3 OFMAP memory

The OFMAP memory is embedded within the Adder Tree Block, it serves as an intermediate storage for the partial-sums until the final ofmap values have been computed. Due to the nature of the accelerator, each processing pass computes r primitive ofmap channels belonging to the same ofmap channel, hence they can be added together in the Adder Tree. The further addition that can be done after the computation within the PE Array is given by the number of PE Sets (r). At the same time, the spatial mapping parameter p allows reusing the ifmap pixels by computing different ofmap channels sequentially. While this allows the accelerator to reuse the ifmap pixels, this comes at the cost of having to temporarily allocate the M ofmap channels in order to accumulate those values. Because of this, in order to not waste any clock cycle while loading the previously stored value, the accelerator uses double-port SRAM.

On the other hand, the bias is also added within the OFMAP memory. The bias is sent directly from the Weight memory at the appropriate time. Both the accumulation of the ofmap primitives and the bias addition takes place in the Front-End Accumulation Interface, shown in Figure 3.7. When the signal NoC_c (index coming from the *OFMAP NESTED LOOP*) is zero, there is no need to read other ofmap values since it is yet the first iteration and there is nothing to accumulate, but once NoC_c is no longer zero then accumulation is enabled (first multiplexer) and the Front-End Acc. Interface sets the control signals for the Back-End interface to enable the memory reading the corresponding ofmap value. Similarly, the signal NoC_c also sets the condition for adding the bias value (second multiplexer).

A third multiplexer is used to control the bitwidth of the ofmap value when all accumulations have finished, which as defined by parameter $OFMAP\_BITWIDTH$, is 34. Truncation to both the upper bound and lower bound is applied.

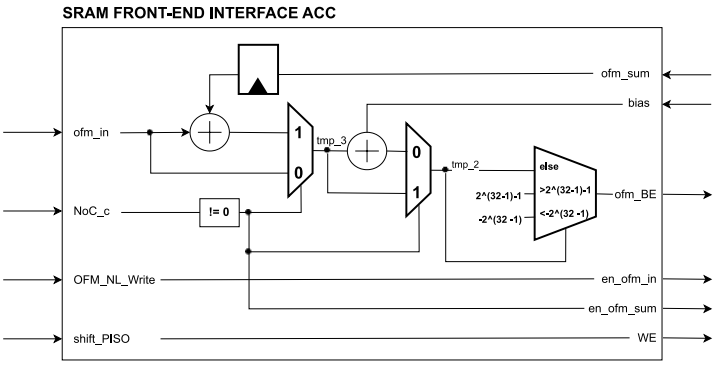


Figure 3.7: Block Diagram of OFMAP memory's Front-End Acc. Interface.

Since the most suitable available double-port compiled memory from *STM* for this purpose was the DPHD 2048x32, this one was used, and a router was implemented to interact with the Back-End Interface. The Front-End Out Interface reads the final values and outputs them to the ReLU/RN block as soon as the final ofmap values are obtained. The Front-End Out interface it is very simple since the order at which the values are output corresponds to the order at which they must be in the Activation memory (column-wise, row-wise and channel-wise), see Figure 3.8. A high-level block diagram is depicted in Figure 3.9, showing all the interfaces.

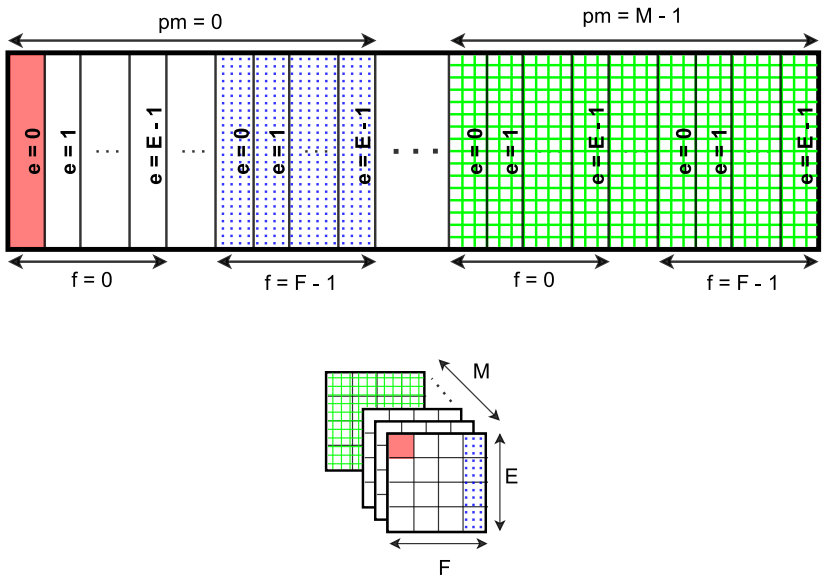


Figure 3.8: Memory Mapping of ofmap values in OFMAP memory.

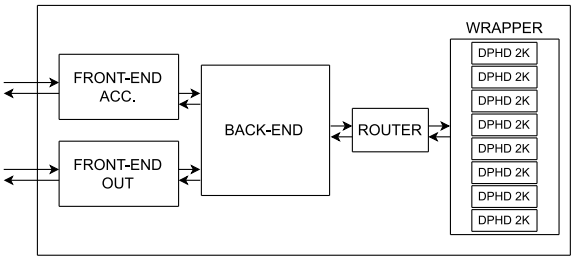


Figure 3.9: High level Block Diagram of OFMAP memory.

3.6 Network-on-Chip

The NoC facilitates communication between the PE Array and the various SRAMs that compose the on-chip memory, as well as between the individual PEs. On the one hand, it connects the Weight memory and Activation memory with the PEs, prior pass through the MCs to evaluate to which PE the input values must go. On the other hand, it connects the PEs themselves vertically to accumulate the psums.

The NoC must possess the flexibility to accommodate various data distribution patterns, originating from the diverse configurations of the CNN and the sequence of operations. Furthermore, it manages the three parameters involved in the convolution; the ifmap pixels, weight values, and psums. The latter becomes an ofmap value after the necessary accumulations have been performed. Each data type has their own separate buses to write/read these parameters to/from their corresponding SRAM modules or from the PE Array, concurrently enhancing the on-chip memory bandwidth.

In an effort to minimize the hardware resources used during implementation, the dimensions of the PE Array are set to a minimum to ensure the operation of the largest convolutional layer of the Baseline CNN Model, avoiding the need of *folding* the convolution, hence the reason of the PE Array of having a size of *32x3*.

The NoC engine is configured based on the hardware parameters and the configuration parameters, and it is also set as to recognize which layer is being computed; it recognizes dimensions of the ifmap, ofmap and filter and hence it knows:

- Which PEs must receive which ifmap/weight.
- Number of PE Sets and Dimensions of the PE Set.

PE Set dimensions and number of PE Sets are not hardwired, but the control flow will decide depending on network characteristics. In the following subsections the blocks that conform the NoC as well as their arrangement are described.

3.6.1 Multicast Controllers

The Multicast Controllers (MCs) are a key element in the NoC, their function is to control the data flow of the incoming input data from the Activation memory and the Weight memory to the PEs.

Each PE is linked with an MC, and each one has a hardwired ID. In the case of the activations, the NoC fetches the activation pixel from memory to each PE trough the dedicated bus, then a MC associated to a specific PE decides whether to

disregard the activation pixel sent or not. The weights have a different dedicated bus. Both buses work in parallel.

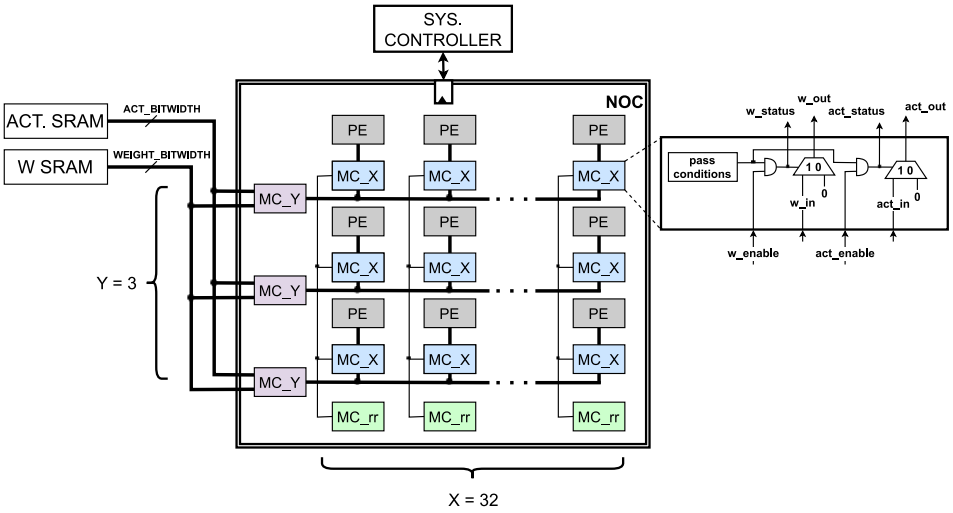


Figure 3.10: High level Block Diagram of NoC Interface, with a close up look to the internal structure of the Multicast Controller.

This work uses a bus structure (see Figure 3.10) with X vertical Y-buses connected to Y horizontal X-buses. Each of the Y horizontal X-buses have a different Y-ID, each PE within a row has an X-ID and a Y-ID associated to its X-bus. Each node within the Y-X network has an MC. Hence, so far:

- The Y X-buses are attached to the Y-bus, and each of these X-buses have X associated MCs and X PEs to which the MCs are linked to, with their corresponding Y-ID and X-ID. For example, the leftmost PE at the top of the PE Array and its associated MC_X have $X\_ID = 0$ and $Y\_ID = 0$, whereas the rightmost PE at the bottom of the PE Array and its associated MC_X have $X\_ID = X - 1$ and $Y\_ID = Y - 1$
- There are two separate buses for both activations and weights to provide sufficient bandwidth from the memories to the PE Array.
- The buses are *ACT BITWIDTH* and *WEIGHT BITWIDTH* bits wide for activations and weights, respectively. Logic within the MC allows the input data to be sent to their corresponding PEs in a single read operation.

A closer look to the internal structure of the MCs shows that, in order to let the values pass further through the NoC, the pass conditions and the enable signals must be set to 1, otherwise the value passing is a zero and the status signal is 0, hence ultimately not enabling the PE to which the MC is connected.

A third special type of blocks are the MC_rr, which are connected vertically to all MC_X blocks of each column in the PE Array. The function of the MC_rr

block is to feed to the MC_x blocks of each column with the parameter rr, which states to which PE Set does the PEs of a specific column belong to, depending on the characteristics of the convolutional layer being computed. For example, if there are $r = 4$ PE Sets, the 9_{th} column belongs to the second PE Set ($rr = 2$ for that column). On the other hand, if there are $r = 1$ PE Sets then that column belongs to the 1_{st} (and only) PE Set ($rr = 1$).

The order at which the activations are sent from the memory to the MCs is column-wise. The number of clock cycles it takes to distribute the activations across the PE Array is $W' \cdot H' \cdot r$ clock cycles. Table 3.11 describes how many clock cycles it takes to distribute the activations on each convolutional layer of our Baseline CNN Model. In a similar way, the number of clock cycles that takes for the weights to be distributed across the PE Array is given by Table 3.12. Weights are distributed horizontally, meaning each row of PEs in the PE Set allocates p different kernel's rows belonging to the corresponding ifmap channel being computed on each PE Set.

Table 3.11: Arrangement of Activations across the PE Array.

	# of cc per PE Set $(W' \cdot H')$	**# of cc in total** $(W' \cdot H' \cdot r)$
Conv. 0	1156	1156
Conv. 1	1156	1156
Conv. 2	324	648
Conv. 3	324	648
Conv. 4	100	400
Conv. 5	100	400

Table 3.12: Arrangement of Weights across the PE Array.

	# of cc per PE Set $(R \cdot S \cdot p)$	**# of cc in total** $(R \cdot S \cdot p \cdot r)$
Conv. 0	72	72
Conv. 1	72	72
Conv. 2	72	144
Conv. 3	72	144
Conv. 4	72	288
Conv. 5	72	288

The MC must be able to decide if the input value shall pass or not by using hyperparameters of the network. There are several *pass conditions* which are tailored for the MC_X and MC_Y blocks an which are described below.

- **Activations *pass conditions***
  - 1^{st} condition: determines the upper bound for which the activation value shall no longer pass to the top rows in the PE Array.

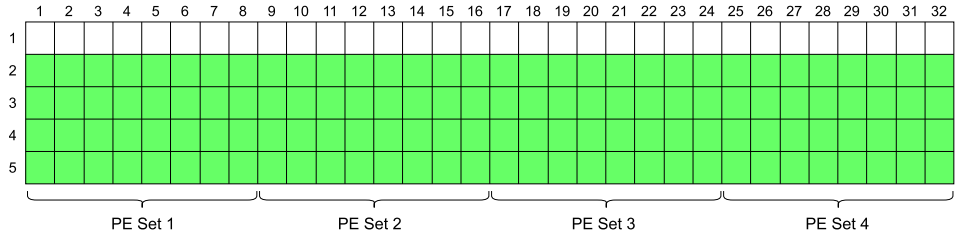
  - 2^{nd} condition: determines to which PEs the activation must go. The condition takes into account how many PE Sets there are and to which PE Set the specific PE belongs to. It allows passing the value to several PEs diagonally.
  - 3^{rd} condition: determines to which ifmap channel the activation belongs to.
  - 4^{th} condition: determines the lower bound for which the activation value shall no longer pass to the bottom rows in the PE Array.

To exemplify, let's take Conv. 5 layer of our Baseline CNN Model over a PE Array with $X = 32$ and $Y = 5$. Conv. 5 has an ofmap channel with height/width of 8 ($EF = 8$), an ifmap channel, including padding, with height/width of 10 ($H'W' = 10$) and 64 ifmap channels ($C = 64$). Because $EF = 8$, the PE Array can allocate 4 PE Sets ($r = 4$).

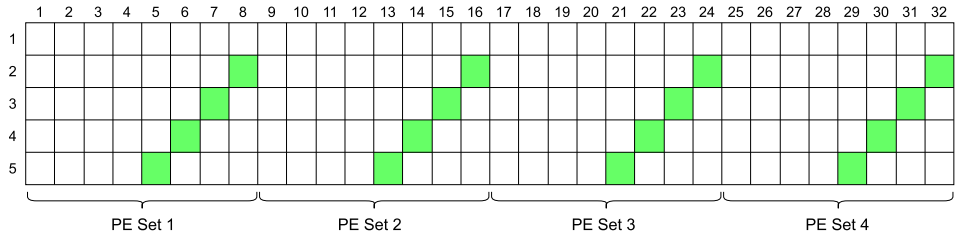
Supposing the eighth row ($h' = 8$) of the fifteenth ifmap channel ($c = 15$) is sent to the PE Array:

- 1^{st} condition is satisfied for the last four rows as per Figure 3.11a.
- 2^{nd} condition is satisfied for the PEs in diagonal green shown in Figure 3.11b.
- 3^{rd} condition states that the pixel must be sent to the last PE Set in the PE Array (Figure 3.11c).
- 4^{th} condition disregards the last two rows in the PE Array, because we are using a 3 x 3 kernel (Figure 3.11d).

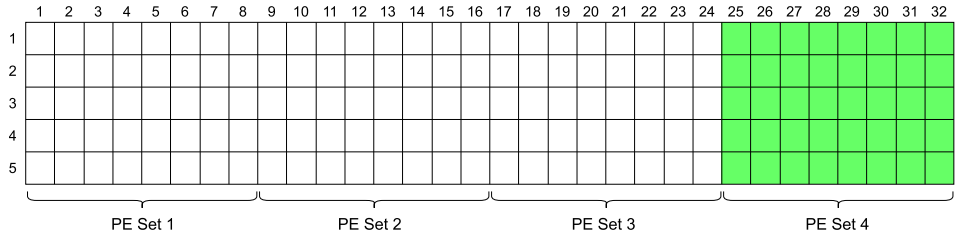
As a result, the PEs to which the activation value for the indices $h' = 8$ and $c = 15$ must be sent are shown in Figure 3.11e.



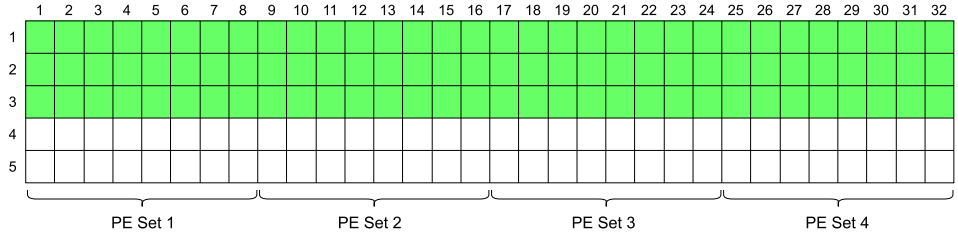
(a) 1^{st} pass condition for Activations



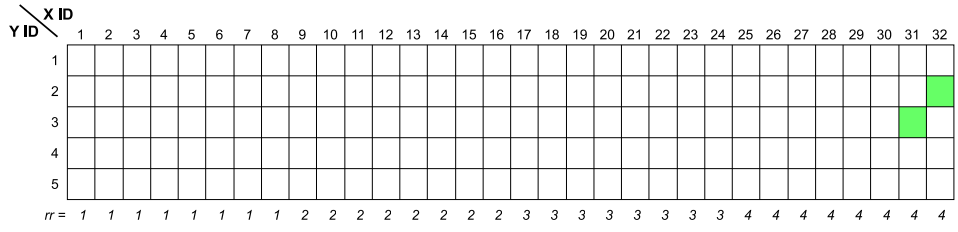
(b) 2^{nd} pass condition for Activations



(c) 3^{rd} pass condition for Activations



(d) 4^{th} pass condition for Activations

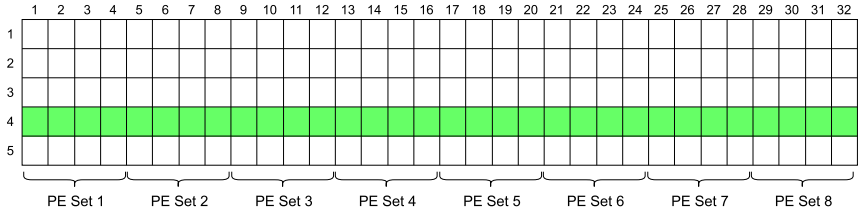


(e) Resulting PEs selected

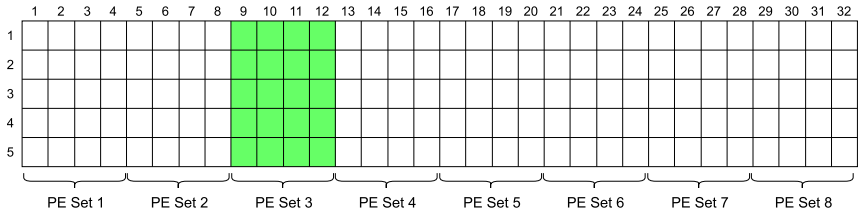
Figure 3.11: Pass conditions and resulting PEs selected on a PE Array with $X = 5$ and $Y = 32$ for Conv. 5 layer of the Baseline CNN Model for activation value with indices $h' = 8$ and $c = 15$.

- **Weights *pass conditions***
  - 1^{st} condition: determines the row of the PE Array.
  - 2^{nd} condition: determines to which ifmap channel the weight belongs to. It is the same as the 3^{rd} condition for the activation values.

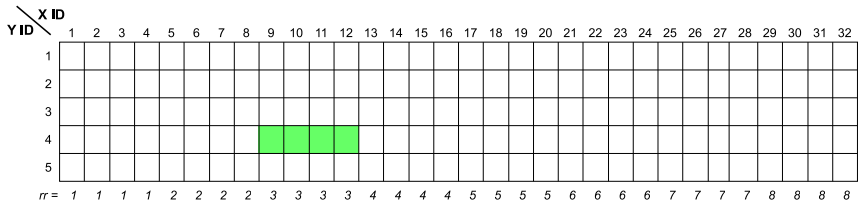
To exemplify the flexibility of the accelerator, suppose a convolutional layer with a 5 x 5 filter, 16 ifmap channels ($C = 16$) and an ofmap channel with height/width of 4 ($EF = 4$). Supposing the weight value to be sent to the PE Array belongs to the second ifmap channel ($c = 2$) and to the third row of the filter kernel ($r' = 3$); The PE Array is now divided in 8 PE Sets ($r = X/E = 32/4 = 8$). The selected PEs to which the weight must be sent are shown in Figure 3.12.



(a) 1^{st} pass condition for Weights



(b) 2^{nd} pass condition for Weights



(c) Resulting PEs selected

Figure 3.12: Pass conditions and PEs selected on a PE Array with $X = 5$, $Y = 32$ for a conv. layer with a 5 x 5 kernel, $C = 16$, $E = 4$. The indices of the weight are $c = 2$, $r' = 3$.

In conclusion, the described approach allows the accelerator to send a single input value to their corresponding PEs in a single read operation, alleviating on-chip memory bandwidth.

3.6.2 Processing Elements

The Processing Elements (PEs) are the core units of the accelerator. The PE has a multiplier unit and an adder to multiply and accumulate the input data and generate the psums. It also has two internal Register Files (RFs) for storing the activations and weights that are read from the memories. There are 2 levels of memory hierarchy (on-chip SRAM and RFs) in the accelerator, being the deeper one composed by the local RFs, allowing an NDP approach and thus improving overall energy efficiency and optimizing data reuse. During a processing pass, when activations and weights have been casted from the memories, each PE in the PE Array holds the input data needed as per RS dataflow, hence all PEs in play can start computation concurrently. The overall architecture of the PE is shown in Figure 3.13.

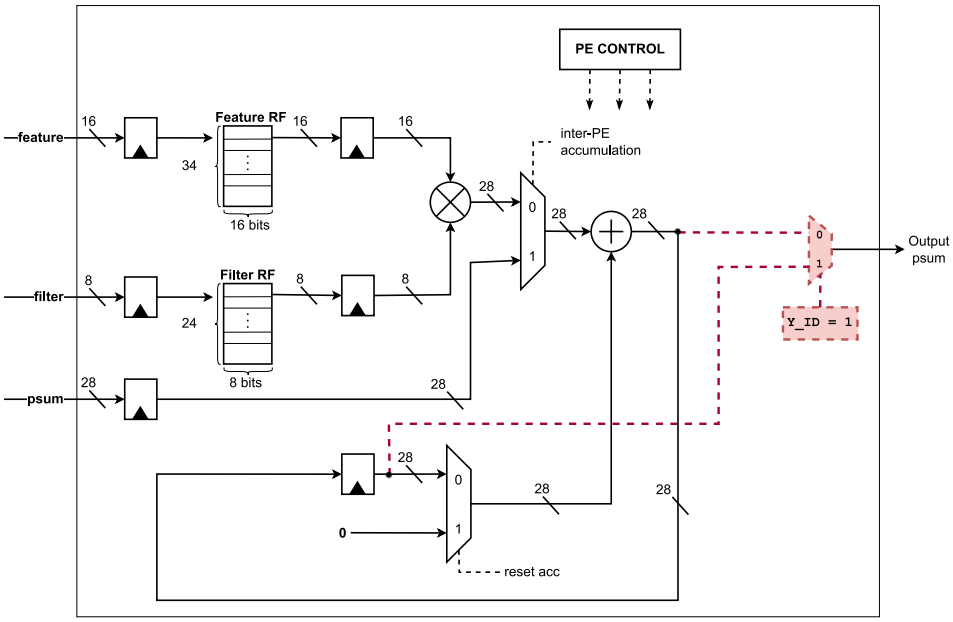


Figure 3.13: Block Diagram of a Processing Element.

The computation starts when a control signal coming from the system controller states that all input data necessary for the current processing pass have been already sent to the RFs. At this point, in the *Feature RF* there is one row of activations, and in the *Filter RF* there are p rows of weights. Taking into account that psums are accumulated vertically, all PEs within a PE column in the PE Array will first compute internally S MACs, and the psums obtained internally within each PE in the PE Column will be sent to the top PE in the PE Column, where the psums will be further accumulated, computing R MACs. hence, it takes $R \cdot S$ MACs to obtain an ofmap primitive and, as seen in Figure 3.14, it takes $R + S$ clock cycles to obtain the ofmap primitive that is later to be sent to the Adder Tree. Each PE Column computes a different ofmap primitive concurrently, thus on each iteration there are $E \cdot r$ ofmap primitives.

Intra-PE accumulation operation takes place first, the first activation value and the first weight of the first kernel row are read and sent to the multiplier unit, then the second activation is multiplied by the second weight and accumulated with the previous operation, this is repeated S times. Then *Inter-PE Accumulation* proceed, and the computed psums are transferred upwards in the PE Column and accumulated in the top PE. When all MACs for obtaining the ofmap primitive have been computed, the ofmap primitive is stored in the accumulator register of the top PE and sent to the Adder Tree. Right after, that register is reset in order to accumulate the next ofmap primitive. While *Inter-PE Accumulation* takes places in the top PE, the other PEs in the PE Column await. The logic dashed in Figure 3.14 indicates the extra logic implemented exclusively for the top PE in order to perform the extra accumulation, this logic is not implemented in the other PEs.

When the operation described is finished, there are X $(E{\cdot}r)$ ofmap primitives in each of the top PEs of the PE Array that, in the case when there are more than one PE Set, need to be accumulated further and hence they are sent to the Adder Tree, where this extra accumulation proceed. Afterwards, they are sent to a Parallel-In Serial-Out buffer (PISO Buffer) prior to be stored into the OFMAP memory, where subsequent ofmap primitives are accumulated until obtaining the final ofmap value. When the PISO Buffer is full and while it is being emptied to the OFMAP memory, the computation in the PE Array must stall and wait for the PISO Buffer to be empty before continuing the operation. The computation happening in the PE Array, the Adder Tree, the PISO Buffer, and the OFMAP memory during the write operation is pipelined to minimize this *stall*. Such stalling will depend on the number of the PE Sets of the convolutional layer being computed. The specific *stall time* will be described when the architecture of the Adder Tree and the PISO Buffer are presented. Ideally, without accounting with the stall time, the *Intra-PE Accumulation* and *Inter-PE Accumulation* is repeated $F \cdot p$ times until a new pass starts.

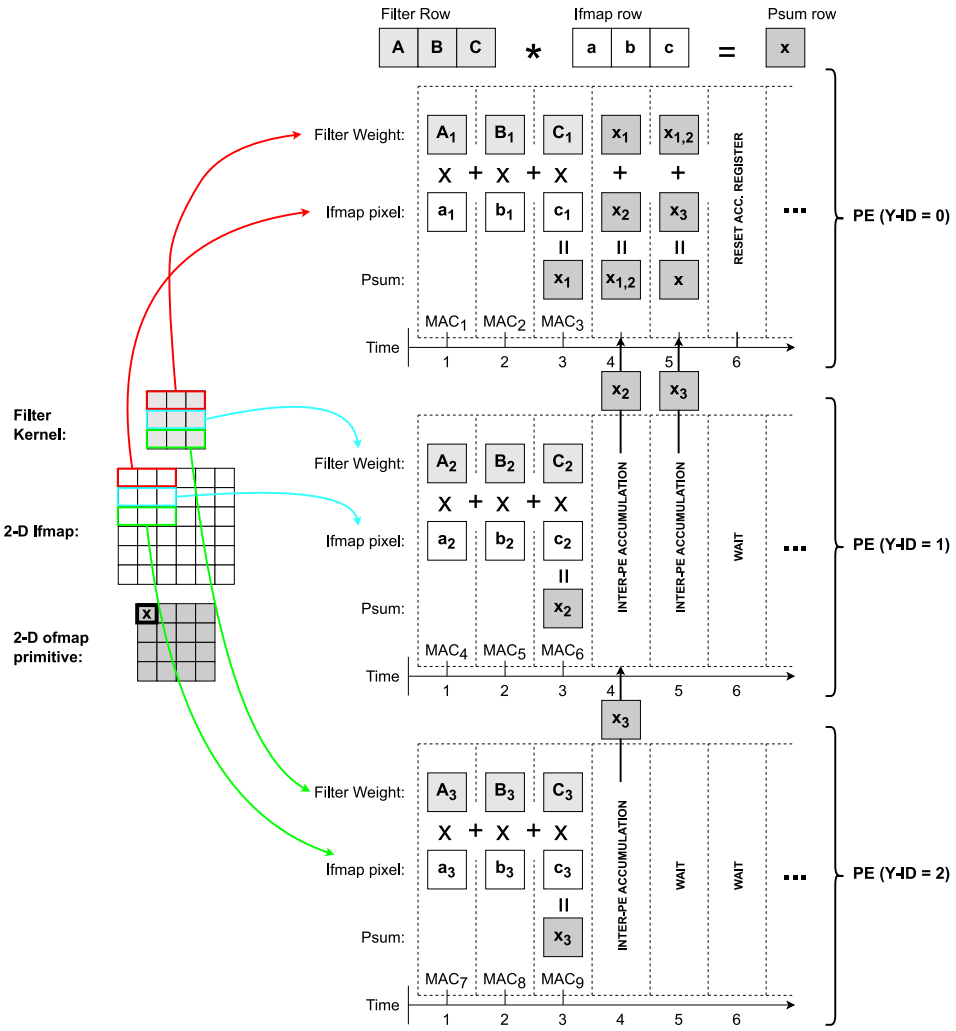


Figure 3.14: *Intra-PE Accumulation* and *Inter-PE Accumulation* operation.

3.7 Adder Tree

The Adder Tree fetches $E \cdot r$ ofmap primitives from the PE Array and performs further accumulation before sending the resulting E ofmap primitives to the PISO Buffer. The Adder Tree must be flexible enough to reroute the ofmap primitives according to the number of PE Sets being computed. If there is only one PE Set, then there is no further accumulation. However, if $r = 2(4)$, this means that $E = 16(32)$. Each ofmap primitive from $e = 0$ to $e = E - 1$ of each PE Set must be added accordingly. Figure 3.15 shows the structure of the Adder Tree. Given that the accelerator allows the PE Array to be divided of up to four PE Sets, the Adder Tree must be able to handle this borderline scenario, so that on its first computing stage it first evaluates whether $r = 1$ or not, in which case there is no need to add anything, and the X ofmap primitives can be sent to the PISO Buffer. If $r \neq 1$, the X ofmap primitives are split in four buses of size $X/4$. At this point, notice that, regardless of $r = 2$ or $r = 4$, the first bus and the third bus (dashed lines) contain ofmap primitives that are to be added together, and the second bus and fourth bus (dotted lines) contain ofmap primitives that are to be added together.

For example, let $r = 2$, then there are two PE Sets, and each one computes ofmap primitives from $e = 0$ to $e = X/2 - 1$. The Adder Tree must add together each pair of ofmap primitives. During the first stage, one adder adds the pairs of ofmap primitives from $e = 0$ to e $= X/4 - 1$ (dashed lines). The other adder adds the pair of ofmap primitives from $e = X/4$ to $e = X/2 - 1$ (dotted lines). Each adder sends the resulting $X/4$ ofmap primitive to their corresponding registers at the next stage, and then a multiplexer evaluates if $r = 2$ or $r = 4$. Since $r = 2$, the buses are joint together obtaining the $E = X/2$ ofmap primitives, which are then sent to the PISO Buffer. If, on the other hand, $r = 4$, the ofmap primitives are further added altogether in the second stage, obtaining a bus composed by $X/4$ ofmap primitives which are sent to the PISO Buffer.

As it can be seen, the number of clock cycles that takes to the ofmap primitives going through the Adder Tree depends on the number of PE Sets and thus the number of additions it needs. Hence, it can be stated that $\#\ cc\ Adder\ Tree = log_2(r)$. A control signal coming from the PE Array is also delayed the same number of cycles to synchronize the dataflow with the control of the PISO Buffer. It is worth mentioning that, depending on the number of PE Sets, the bus width to be sent to the PISO Buffer differs, so when $r = 1$ there are X ofmap primitives to feed the PISO Buffer with, when $r = 2$ the size of the bus is $X/2$, and when $r = 4$, the size of the bus is $X/4$.

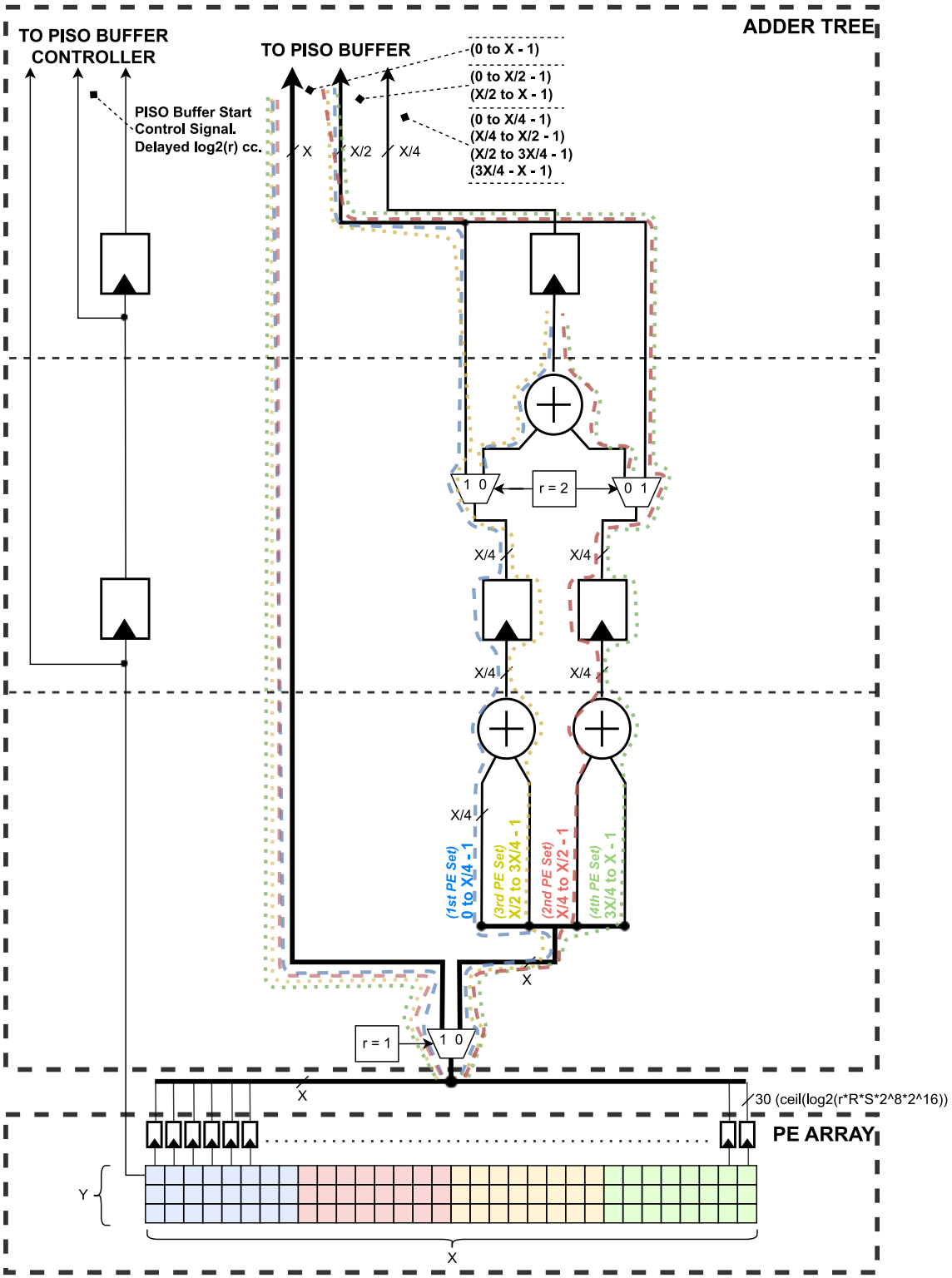


Figure 3.15: Adder Tree structure.

3.8 PISO Buffer

The Parallel-In Serial-Out (PISO) Buffer is composed by X registers. Depending on the numbers of PE Sets r, the batch size of ofmap primitives that the PISO Buffer receives from the Adder Tree on each iteration will vary.

If $r = 1$, the batch has a size of X hence every time the PE Array computes the E ofmap primitives and these go through the Adder Tree all the way to the PISO Buffer, this gets already full and needs to be emptied towards the OFMAP memory. In the meantime, the operation in the PE Array must stall and wait for the PISO Buffer to be empty. When $r = 2$, the batch has a size of $X/2$ hence the PE Array operation can proceed two consecutive times before the PISO Buffer is full, and then it has to wait $X/2$ clock cycles until there is enough space in the PISO Buffer for another batch. When $r = 4$, the PE Array operation iterates four times until the PISO Buffer is full, and $X/4$ clock cycles must pass before allowing sending another batch of ofmap primitives.

The control of the PISO Buffer allows sending each batch to the corresponding section in the buffer, so when $r = 4$ and if the PISO Buffer is empty, the batch is send to the rightmost section in Figure 3.16, on the second iteration the batch goes to the second section, and so on until the buffer is full. This operation proceeds continuously until the processing pass finishes, after which the buffer is emptied.

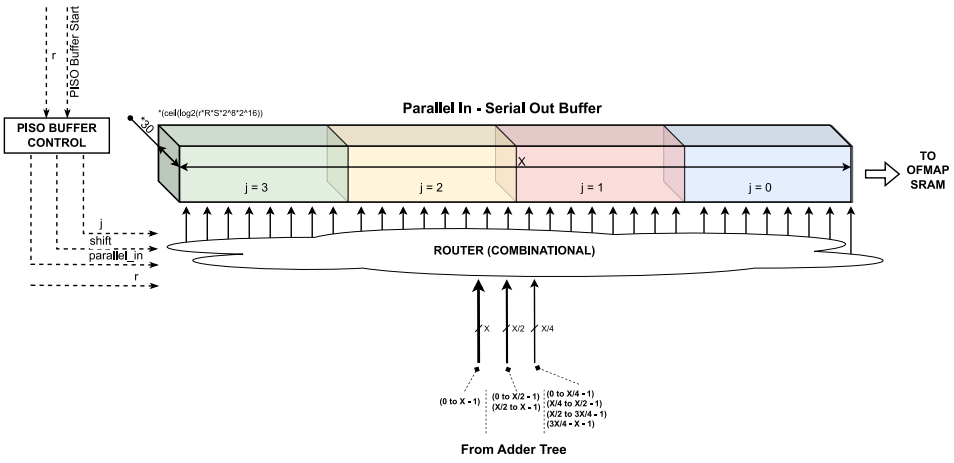


Figure 3.16: PISO Buffer structure. There are four sections for allocating different batches of ofmap primitives.

The *stalling time* that the PE Array must wait before proceeding with the computation of the next batch of E ofmap primitives will depend on the number of PE Sets r, which ultimately depends on the size of the batch E ($r = X/E$), and the size of the kernel (R x S): $stall\_time = E - R - S + 1$.

3.9 ReLU & Rounding

Once all processing passes for the current convolutional layer being computed have finished, the values in the OFMAP memory are ready to be sent back to the Activation memory. Prior to that, these values must be rounded down to 16 bits and ReLU activation function must be applied. ReLU activation function is straightforward and the logic evaluates whether the ofmap value is higher than zero, in which case it let the value pass, otherwise it returns zero.

Round To Nearest (RN) is applied according to Section 3.11. The ofmap value at this point is a 32 bits fixed-point number with 19 bits belonging to the integer part and 13 fractional bits ($Q < I.F >= 32 < 19.13 >$). The activation is a 16 bits fixed-point number with 8 bits belonging to the fractional part and 8 fractional bits ($Q < I.F >= 16 < 8.8 >$). In order to trim the fractional part and apply RN a 1 must be added to the fractional bit at position $13 - 8 = 5$, and disregard the rightmost bits. For truncating the integer part the $19 - 8 = 11$ leftmost bits are disregarded, obtaining 8 bits. There is no need to handle the sign bit since all truncation is applied after performing ReLU hence all values are positive.

3.10 Pooling

When the configuration parameter *is_pooling* is 1, before storing the activations into Activation memory, max-pooling is applied. The activations arrive to the Pooling Block column-wise (from $e = 0$ to $e = E - 1$), row-wise (from $f = 0$ to $f = F - 1$, and channel-wise (from $m = 0$ to $m = M - 1$), and they are stored in the Activation memory in the same order.

A buffer of size $X/2$ is used to be able to allocate the biggest possible size given when there is only one PE Set ($r = X/E = 32/32 = 1$). At first, a multiplexer sends the first value from the first column being computed to *register 1*, and the next value to *register 2*, both values are compared and the result is stored in the buffer, thus when processing the entire column half of it will be allocated in the buffer with the results of the first comparison.

Once the first half column is in the buffer, the first and second activations from next column are compared, and the result is sent to *register 3* while the first value of the previously processed column is read from the buffer. Both values are then compared and the bigger of these two values is then the biggest of the four activations. This process is repeated until the second column is processed, then the third column overwrites the contents in the buffer and when the fourth column starts the third column in the buffer is compared with the incoming fourth column. The operation proceeds until reaching the last column ($e = E - 1$).

There is no any extra cycle as a consequence of this logic and the time that it takes to write the activations back to the Activation memory is given by its size $(E \cdot F \cdot M)$, regardless of max-pooling being applied or not. Figure 3.17 shows the block diagram of the Pooling block and exemplify this process.

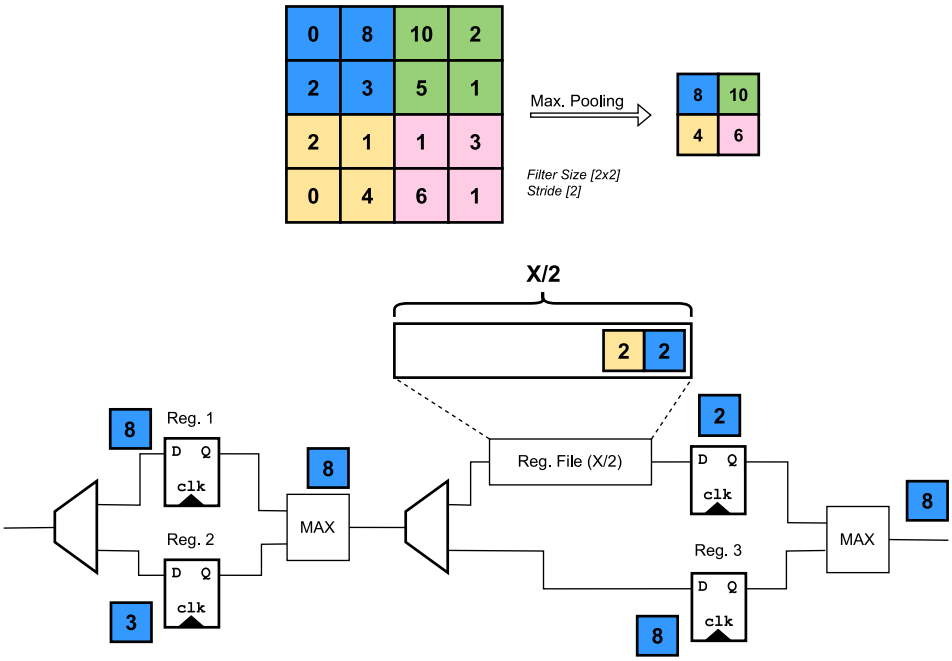


Figure 3.17: max-pooling Block Diagram.

3.11 Quantization and Rounding Scheme

Even though the hardware parameters of the accelerator allows choosing a specific bitwidth for weights, activations and biases, the current work has been analyzed with these values being 8, 16 and 16, respectively. Lower bitwidths degraded achievable accuracy for the Baseline CNN Model.

Furthermore, the position of the decimal point also affects accuracy. The precision of activations and weights affects the overall accuracy of the network's predictions. With a low number of fractional bits, the intermediate values may be rounded or quantized in a way that discards important information, leading to a decrease in the network's accuracy. On the other hand, a low number of integer bits may cause the activations to be rounded to the nearest quantization level, leading to a loss of information that can also impact the performance of the network. Table 3.13 summarizes the quantization applied:

Table 3.13: Quantization Scheme.

	BITWIDTH $(Q < I.F >)$	**Integer Part** (I)	**Fractional Part** (F)
Activations	16	8	8
Weights	8	3	5
Biases	16	3	13
ofmap	34	21	13

The ofmap's bitwidth is given after all MACs for obtaining such value have finished.

Since MAC operations imply multiplications, the fractional and integer parts must be handled appropriately:

$F_{ofmap} = F_w + F_{act} = 5 + 8 = 13$

$I_{ofmap} = Q_{ofmap} - F_{act} = 34 - 13 = 21$

The bitwidth's data-path of the ofmap primitive increases gradually as it moves from the PE Array to the OFMAP Front-End Acc. interface, where the bias is added to the value before storing it in the OFMAP memory. Note that, since the fractional part of both biases and ofmaps are the same, there is no need to align the decimal point. Also, at this point the ofmap's bitwidth is 34, but the memory's bitwidth is 32, thus truncation to max/min value is applied and the 2 MSB's are disregarded, this is done in hardware while also taking into account the sign bit.

3.12 Wrapping Up

Once described the whole architecture of the accelerator, it is possible to calculate the number of clock cycles that it takes to compute a convolutional layer as a function of its parameters. Table 3.14 details the number of clock cycles that each layer of the Baseline CNN model needs for it to distribute the activations and weights to the PE Array before actual computation starts. Note that both weights and activations are sent concurrently to the PE Array, and note also that there are some passes in which only weights are loaded while activations are still being reused.

Table 3.15 describes the number of clock cycles for computing the ofmaps in the PE Array, the Adder Tree and the PISO Buffer on their way to the OFMAP memory. Table 3.16 describes the number of clock cycles for writing the resulting activations back to Activation memory, as well as the time it takes to load the configuration parameters and the time it takes to the main NL updating its values.

Combining the results from previous tables, it is possible to calculate the total number of clock cycles that it takes to the accelerator to process each convolutional layer (see Table 3.17). As it can be seen, this time responds to the characteristics of each layer and how it is mapped in the accelerator, and it can be known in advance. With this data the overall performance of the accelerator is computed in the *Results* 6 chapter.

[1] $stall \cdot F \cdot p - stall \cdot (r-1) = stall \cdot ((F \cdot p) - (r-1)) = (E-R-S+1) \cdot ((F \cdot p) - (r-1))$. Notice that at the beginning of computations, PISO Buffer is empty so PEs do not stall until PISO Buffer is full. PISO Buffer gets full after (. e.g. if $r = 4$, it is possible to load the buffer four times before stalling.

[2] RN, ReLU and max-pooling takes place as well.

[3] During computation of the first convolutional layer in the CNN the parameters are loaded twice; once at the very beginning for computation of current layer and once at the end for computation of the next layer. During the last convolutional layer, there is no loading of parameters since those are loaded at the end of computation of previous layer.

Table 3.14: # of Clock Cycles for distributing both activations and weights across the PE Array before computation starts.

	Convolutional Layer:					
	0	**1**	**2**	**3**	**4**	**5**
Distribution of Activations (1 pass)						
- Act. Nested Loop Logic (start) (r)	1	1	2	2	4	4
- Switching PE Sets $(r-1)$	0	0	1	1	3	3
- Act. Nested Loop Logic (end) (r)	1	1	2	2	4	4
- Distribution of Acts. $(W' \cdot H' \cdot r)$	1,156	1,156	648	648	400	400
Total	**1,158**	**1,158**	**653**	**653**	**411**	**411**
Distribution of Weights (1 pass)						
- Weights Nested Loop Logic (start) (r)	1	1	2	2	4	4
- Switching PE Sets $(r-1)$	0	0	1	2	3	3
- Weights Nested Loop Logic (end) (r)	1	1	2	2	4	4
- Distribution of Weights $(p \cdot r \cdot R \cdot S)$	72	72	144	144	288	288
Total	**74**	**74**	**149**	**149**	**299**	**299**
Distribution of Activations (total)						
- Act. Nested Loop Logic (start) $(r \cdot Passes)$	6	32	64	128	256	512
- Switching PE Sets $(r-1) \cdot Passes$	0	0	32	64	192	384
- Act. Nested Loop Logic (end) $(r \cdot Passes)$	6	32	64	128	256	512
- Distribution of Acts. $(W' \cdot H' \cdot r \cdot Passes)$	6,936	36,992	20,736	41,472	25,600	51,200
Total	**6,948**	**37,056**	**20,896**	**41,792**	**26,304**	**52,608**
Distribution of Weights (total)						
- Weights Nested Loop Logic (start) $(r \cdot Passes)$	6	32	64	128	256	512
- Switching PE Sets $(r-1) \cdot Passes$	0	0	32	64	192	384
- Weights Nested Loop Logic (end) $(r \cdot Passes)$	6	32	64	128	256	512
- Distribution of Weights $(p \cdot r \cdot S \cdot S \cdot Passes)$	432	2,304	4,608	9,216	18,432	36,864
Total	**444**	**2,368**	**4,768**	**9,536**	**19,136**	**38,272**
Distribution of Weights & Activations combined (total) $((C/r) \cdot \#cc\ Acts.) + ((Passes - (C/r)) \cdot \#cc\ Weights)$						
Total	**3,696**	**19,712**	**8,800**	**17,600**	**20,032**	**40,064**

Table 3.15: # of Clock Cycles for computing the ofmap values.

	Convolutional Layer:					
	0	**1**	**2**	**3**	**4**	**5**
Computation of ofmaps (1 pass)						
- OFMAP Nested Loop Logic (start) $(1cc)$	1	1	1	1	1	1
- Intra PE Acc. + Inter PE Acc. $(S+R) \cdot F \cdot p$	1,536	1,536	768	768	384	384
- Stalling[1] $(stall \cdot (F \cdot p \cdot (r-1)))$	6,912	6,912	1,397	1,397	183	183
- RF/Adder Tree/PISO Buffer $(2 + log_2(r) + 1)$	3	3	4	4	5	5
- Emptying PISO Buffer $(X - stall)$	5	5	21	21	29	29
- OFMAP Nested Loop Logic (end) $(1cc)$	1	1	1	1	1	1
Total	**8,458**	**8,458**	**2,192**	**2,192**	**603**	**603**
Computation of ofmaps (total) $(... \cdot Passes)$						
Total	**50,748**	**270,656**	**70,144**	**140,288**	**38,592**	**77,184**

Table 3.16: # of Clock Cycles for writing back to Activation memory, for loading configuration parameters and for updating main Nested Loop values.

	Convolutional Layer:					
	0	**1**	**2**	**3**	**4**	**5**
Writing back to Activation memory[2] $(E \cdot F \cdot M)$						
Total	**16,384**	**16,384**	**8,192**	**8,192**	**4,096**	**4,096**
Loading CFG. Params.						
- CFG. Control $(1cc)$	1	1	1	1	1	1
- Updating CFG. Params.[3] $(\#\ Parameters - 1)$	25	12	12	12	12	0
Total	**26**	**13**	**13**	**13**	**13**	**13**
Main Nested Loop Logic $(\#\ Passes)$						
Total	**6**	32	**32**	**64**	**64**	**128**

Table 3.17: # of Clock Cycles that it takes to the accelerator to process each convolutional layer.

	Convolutional Layer:					
	0	**1**	**2**	**3**	**4**	**5**
Total	**70,860**	**306,797**	**87,181**	**166,157**	**62,797**	**121,473**

Chapter 4

Baseline CNN Model

The Baseline CNN Model used for testing and validating the accelerator was chosen based on two criteria. On the one hand, having a somewhat complex and deep enough CNN so that it can perform good enough and can obtain an acceptable validation accuracy when classifying the *CIFAR-10* Dataset [25]. *CIFAR-10* is a dataset composed of 50,000 training images and 10,000 validation images divided in 10 different classes (airplane, truck, frog, cat, dog, deer, automobile, horse, bird and ship). The images have three channels (Red-Green-Blue) and have a size of 32 x 32 pixels.

On the other hand, state-of-the-art CNNs such as the one in [26], which achieves an accuracy of 98.9% in *CIFAR-10*, are usually composed by millions of parameters. Aiming to design an accelerator that can compute CNNs but also considering this work as a starting point towards a more complex architecture (e.g. handle scenarios such as folding layers with bigger dimensions, multiple batches, or that can load different sets of weights per layer), a simpler CNN model was selected. In this chapter the process for selecting such CNN Model is described.

There are several reasons why deeper layers in CNNs are often constructed with smaller spatial dimensions and a greater number of channels.

- Smaller spatial dimensions can help reduce the number of parameters and computation needed to train the network. This is particularly important for large-scale applications where efficiency is critical, as in NNs tailored for edge devices [27].
- Deeper layers with more channels can help the network learn more complex features. As the network goes deeper, it can learn to detect more complex patterns and relationships in the input data, and having more channels allows the network to represent more diverse and subtle features [28].

A tailored CNN, described in Table 4.1 below, was designed in *Python* using *Tensorflow*, an open-source library developed by *Google* for deep learning applications.

Table 4.1: Baseline CNN Model.

Layer	Output Shape	Param #
Conv. 0	(32, 32, 16)	448
Conv. 1	(32, 32, 16)	2,320
max-pooling 0	(16, 16, 16)	0
Conv. 2	(16, 16, 32)	4,640
Conv. 3	(16, 16, 32)	9,248
max-pooling 1	(8, 8, 32)	0
Conv. 4	(8, 8, 64)	18,496
Conv. 5	(8, 8, 64)	36,928
max-pooling 2	(4, 4, 64)	0
Flatten	(1,024)	0
Dense Layer	(10)	10,250
Total # of Parameters:		**82,330**

The model is composed by six convolutional layers, with inserted max-pooling layers to reduce dimensions of the channel. The last layer is a FC layer with ten output neurons which hold the final values corresponding to each one of the classes of the *CIFAR-10* dataset. Overall, the CNN has 82,330 parameters. The network was trained using a batch size of 1, 25 epochs and using stochastic gradient descent (SGD) with momentum. The basic idea behind the momentum method is to add a momentum term to the update rule for the weights, which takes into account the previous weight updates [29].

After training, a validation accuracy of 71% is achieved (see Figure 4.1).

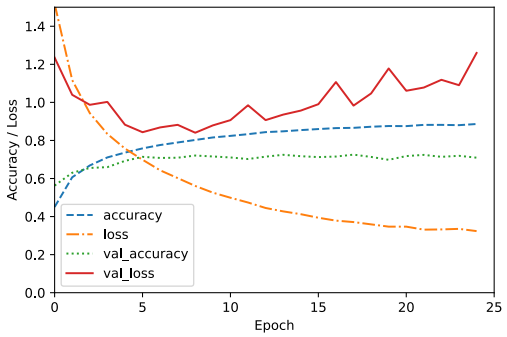


Figure 4.1: History Diagram of the training process of the Baseline CNN Model using SGD and momentum, 25 epochs and no data augmentation.

Notice how the validation accuracy stops improving around the 7th epoch and the validation loss starts increasing. This effect is a sign of *overfitting*, which is a phenomenon where the model performs well on the training data but poorly on new data. There are several workarounds to solve this issue, such as data augmentation. Data augmentation refers to the process of increasing the amount of training data available. This can be achieved by applying a set of transformations to the original images in the training set, such as rotations, translations, flips, and other distortions, to generate new images that are still representative of each class.

By introducing these variations, the model can learn to be more robust to changes in the input data, and can potentially improve its performance. After training again while also increasing the number of epochs to 100, overfitting was reduced and validation accuracy increased up to around 81% (see Figure 4.2.

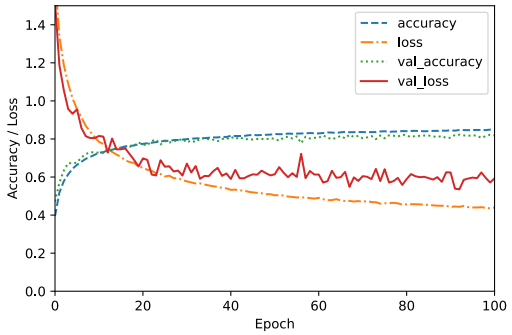


Figure 4.2: History Diagram of the training process of the Baseline CNN Model using SGD and momentum, 100 epochs and data augmentation.

Once the model was trained, the parameters were exported to *MATLAB*. In order to compare the behavior of the actual hardware, a behavioral model was implemented in *MATLAB*, where the parameters of the network are first evaluated in floating point format and analyzed to check their dynamic range for appropriately choosing the position of the decimal point and quantize them accordingly.

MATLAB model also registers the maximum and minimum values of the activations across the network for each forward pass when running the validation dataset. The maximum and minimum activations are 56.554 (dense layer) and -79.870 (Conv.5 layer), respectively. These values determine the dynamic range of the activations, which is the range of values that can be represented by the number format. Hence, at least 8 bits are needed to represent all activations within the range without clipping ($2^{(8-1)} = 128$). A similar analysis was performed regard-

ing the dynamic range of the trained weights. It was observed that an excessive reduction of the precision of the activations and weights resulted in loss of information that degraded the accuracy considerably. Because of this, the most optimal approach for the current model was to work with 16-bit fixed-point for the activations (8 bits for the integer part and 8 bits for the fractional part), and 8-bit fixed point for the weights (3 bits for the integer part and 5 bits for the fractional part).

After running the validation dataset again in *MATLAB* with the mentioned quantization approach, the accuracy achieved was 79.2%. Thus, it can be concluded that the selected model behaves good enough given that it is not as complex as other state-of-the-art CNNs, making it a perfect Baseline CNN Model to evaluate the performance of the accelerator.

4.1 Backpropagation Analysis

Initially, the accelerator was intended to work both for training and inference, so the implemented *MATLAB* behavioral model is also capable of training such described network from scratch. It does so by running backpropagation once the forward propagation of the network is finished. It propagates backwards the computed error at the output and calculates the weight gradients and activation gradients for each layer. Due to the increased complexity of implementing this functionality in hardware, it was disregarded during the design process. In this section, we describe the training analysis of the network and some design considerations of the accelerator that were transferred to the only-inference accelerator.

After the forward pass, the output of the network is compared to the truth label to calculate the loss. Then, the backward pass begins with the calculation of the gradient of the loss function with respect to the output of the network. This gradient is backpropagated through the network layer by layer, calculating the gradient of the loss with respect to each weight in the network. Once the gradients have been calculated, the weights in the network are updated using stochastic gradient descent (SGD) and momentum. The weights are updated in the direction that will decrease the loss, proportional to the magnitude of the gradient and a learning rate hyperparameter. Calculating the gradients of each weight of every convolutional layer is a twofold task.

- First, the weight gradients of the current convolutional layer being processed are computed by convoluting the activations of the current convolutional layer with the activation gradients obtained from the next layer in the network, which act as a filter over the activations of the current layer. The result of the convolution is a 4-D matrix, representing every weight gradient of the convolutional layer.
- Second, the activation gradients of the current convolutional layer being processed are computed by convoluting a flipped version of the weight matrix of this layer with the activation gradients of the next layer in the network,

which act as filter over the flipped version of the weights matrix. The resulting activation gradients are then utilized for computing the activation gradients of the previous layer in the network.

As stated, when all weight gradients are calculated, they are updated using SGD and momentum. This process implies not only an increased computational effort to the accelerator, but also a huge memory overhead, since all the activations produced from the forward propagation need to be re-accessed in the backward training phase for weight gradients computation. In addition, weight gradients of all layers and the activation gradients of the current layer being processed need to be allocated in memory. Also, allocation of the activation gradients for the layer with highest number of activations (conv. 0 and conv. 1 layer) would be needed. The indices of the max-pooling operation would need to be also allocated in order to backpropagate the activation gradients in their correct indices, applying a process called upsampling. The indices of the ReLU activation function would also need to be allocated and the number will depend of the sparsity of each layer.

Furthermore, it is not possible to apply aggressive quantization to the gradients, since trimming the precision bits when applying fixed-point quantization imply a loss of information that would result in a poor performance in finding the optimal weights for finding the best accuracy results. [30] explores several approaches for applying effective quantized training and introduces stochastic rounding, which is a rounding technique that involves randomly rounding numbers to their nearest integer. This is in contrast to deterministic rounding, which always rounds numbers up or down based on a fixed rule. By randomly rounding the gradients during the backward pass, the optimization process can be made more robust to noise and can help prevent the network from getting stuck in local minima. With this approach, [30] quantizes weights, biases, activation gradients and weight gradients to 16 bits, varying the size of the fractional part of the word length, concluding that stochastic rounding is able to preserve the information of the gradients as opposed to round to nearest.

The stochastic rounding algorithm is implemented in hardware in [30] by using a linear feedback shift register (LFSR) to introduce randomness to the value to be rounded, and proves that the hardware overhead of such implementation is minimal. [31] further probes the validity of such rounding scheme by recursively adding up the harmonic series as per Equation 4.1 below.

$$\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \tag{4.1}$$

Figure 4.3 compares the accumulated error of the harmonic summation over 5 million iterations and probes that stochastic rounding outperforms round to nearest approach and *single* floating format.

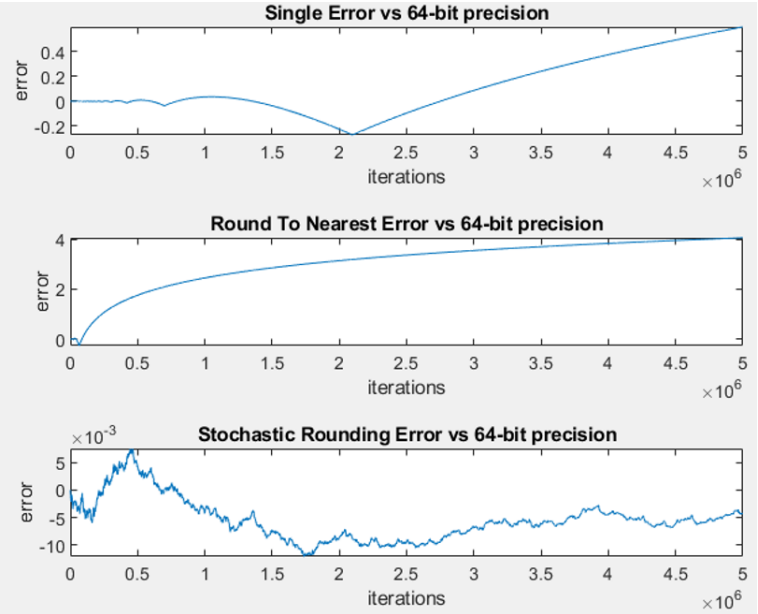


Figure 4.3: Comparison of the accumulated error of different rounding approaches when computing a harmonic series summation over 5 million iterations.

To accelerate the training, [32] introduce stochastic pruning, which is an approach that takes advantage of the fact that activation gradients are mostly composed of values that are very close to zero. They start from the hypothesis that those close-to-zero activation gradients have very little effect to the weight-update process and prune these gradients to zero, demonstrating that it has no effect on the convergence during the training process. In that work, a threshold is chosen based on statistical analysis from which all gradients below this threshold are pruned. By improving the sparsity of activation gradients, this implementation significantly reduces the cost and memory required for the backpropagation procedure during training.

To validate stochastic pruning, we implement in *MATLAB* the training algorithm of a simple CNN composed of one convolutional layer, one max-pooling layer and one dense layer. The CNN is evaluated using MNIST, which is a dataset composed of a large collection of hand-written digits, consisting of a training set of 60,000 examples and a test set of 10,000 examples, where each example is a grayscale image of a handwritten digit (0-9) of size 28x28 pixels. Figure 4.4 shows the normal distribution of the activation gradients belonging to the convo-

lutional layer and exemplifies the stochastic pruning approach as per [32]. Figure 4.5 compares the training efficiency of the *MATLAB* implementation with and without applying stochastic pruning. As it can be seen, there is no noticeable difference in the accuracy achieved.

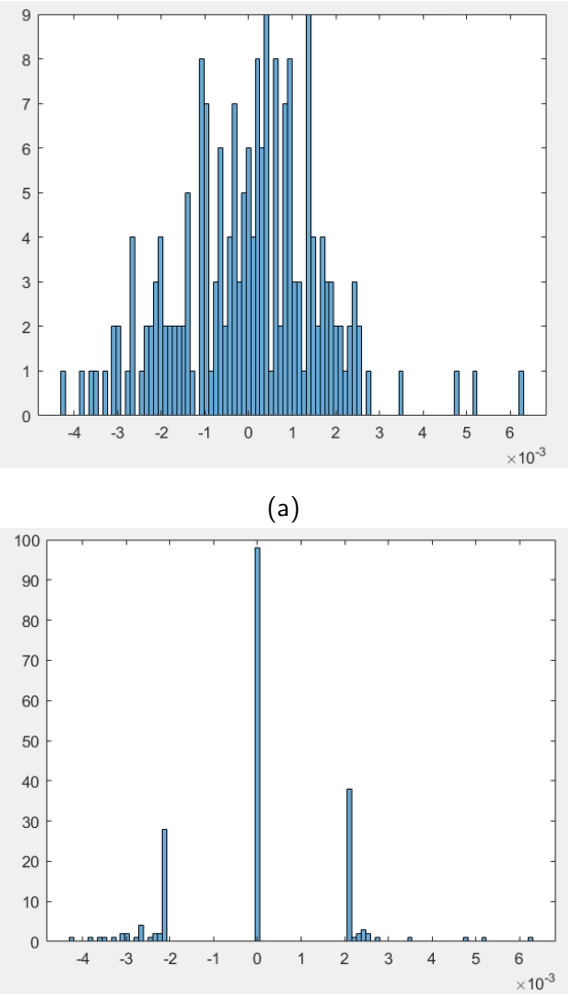


Figure 4.4: Activation gradients distribution before (a) and after (b) applying stochastic pruning.

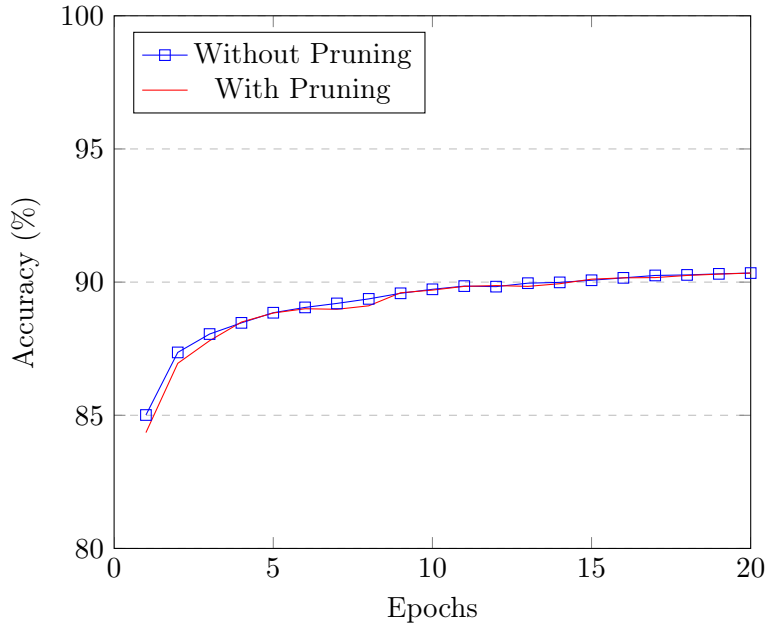


Figure 4.5: Training a CNN for MNIST dataset with and without stochastic pruning.

All in all, implementing training capabilities to the accelerator implies a thoughtful consideration of the memory overhead, the data rearrangement needed for computation of weight gradients and activation gradients, computation of the classification error when forward propagation is finished and the updating of the weight values once the gradients are computed. The present work can serve as a ground line from which these functionalities can be built upon.

Chapter 5
Validation

For validation of the system, different input images from the CIFAR-10 validation set were used as inputs for the accelerator.

Table 5.1: Comparison of the classification quality of the accelerator and MATLAB behavioral model with different images from CIFAR-10 Validation Dataset.

# Of Image	**Real**	**Predicted** *Simulation Behavioral Model*	**Error**
1	Cat	Cat	0.191021
		Cat	0.162236
1923	Dog	Dog	0.000007
		Dog	0.000008
345	Ship	Ship	0.020273
		Ship	0.016767
25	Dog	Deer	1.487420
		Deer	1.330098
5820	Bird	Bird	0.582055
		Bird	0.422870
9582	Horse	Horse	0.000000
		Horse	0.000000
402	Ship	Automobile	2.189124
		Automobile	1.920229
90	Truck	Truck	0.001161
		Truck	0.000959
7026	Frog	Frog	0.000003
		Frog	0.000004
45	Airplane	Airplane	0.047380
		Airplane	0.052249

In order to validate the accelerator, and due to this not being able to compute FC layers, the output values of the max-pooling 2 layer of the Baseline CNN Model are extracted from simulation and fed to the *MATLAB* behavioral model. With the data obtained from the simulation, the *MATLAB* behavioral model processes the dense layer (last layer of our Baseline CNN Model), then applies *softmax* activation function and lastly *cross-entropy* loss function is calculated. The resulting error is compared with the error obtained when entirely running the network in the *MATLAB* behavioral model. Table 5.1 shows such comparison with different images from the validation dataset. It can be seen that the accelerator classify as good as the network does given the trained weights that the CNN works with.

Chapter 6
Results

In this chapter the different metrics of the accelerator in terms of throughput, latency, performance and energy efficiency are presented. The results were obtained by synthesizing the accelerator with the HW parameters from Table 3.6 and Table 3.7.

Throughput, reported in terms of inferences per second, is given by Equation 6.1:

$$throughput = \frac{inferences}{second} = \frac{operations}{second} \cdot \frac{1}{\frac{operations}{inference}} \tag{6.1}$$

Operations per second depends on the frequency, the cycles per operation of a single PE, the number of PEs of the accelerator and the overall time the PE is performing computations, as per Equation 6.2.

$$\frac{operations}{second} = \underbrace{\left(\frac{1}{\frac{cycles}{operation}} \cdot \frac{cycles}{second}\right)}_{\text{for a single PE}} \cdot \#\ PEs \cdot PE\ Utilization \tag{6.2}$$

According to Figure 3.14, it takes 3 clock cycles to compute the expression $A_i \cdot a_i + B_i \cdot b_i + C_i \cdot c_i = x_i$, with $i = 1 \dots 3$ for each PE in a PE Column, plus another 3 clock cycles to add up the psum ($x = x_1 + x_2 + x_3$). For each PE, 3 MACs are computed every 6 clock cycles, being $cycles/operation\ (1PE) = 2$. On the other hand, PE Utilization can be computed as per Equation 6.3 below.

$$PE\ Utilization = \left(\frac{\#Active\ PEs}{\#\ PEs}\right) \cdot Utilization\ Active\ PEs \tag{6.3}$$

The accelerator's architecture focuses on avoiding any idle PE and utilizing all PEs, hence: $\#Active\ PEs/\#\ PEs = 1$.

Considering that the cycles that the PE is actually performing computations is given by $(R + S) \cdot F \cdot p$ for a single pass, in order to calculate the utilization of active PEs we need to multiply this number by the number of passes of each layer, and divide the result by the total number of clock cycles (obtained in Table 3.17). This results in that the PE is performing computations around 25% of the time in which the accelerator is running:

$$\begin{aligned}
&((R+S) \cdot F \cdot p \cdot \#\ Passes)_{\text{Conv. }0} + \\
&\ldots + \\
&((R+S) \cdot F \cdot p \cdot \#\ Passes)_{\text{Conv. }5} = 205,824\ clock\ cycles \\
&205,824/Total\ \#\ clock\ cycles = 205,824/815,365 = 0.252
\end{aligned}$$

The PE Utilization is affected by how quickly data is delivered to the PE without making them idle. This time depends on memory bandwidth and latency, and the data reuse available in the neural network, which is determined by the dataflow and the memory hierarchy. The utilization of PEs can also be affected by an imbalance in work allocated across PEs. This metric could be improved by reducing the stalling time of the PEs or by placing an adder on each PE Column, which would avoid the need of the R extra clock cycles needed for the Inter-PE Accumulation. However, the latter approach would incur a significant hardware overhead. On the other hand, operations per inference from expression 6.1 is given by the total number of clock cycles it takes to compute the whole CNN divided by the batch size, and the batch size that the current version of the accelerator can handle is one. All in all, operations per second can be computed as:

$$\frac{operations}{second} = \underbrace{\left(\frac{1}{2} \cdot frequency\right)}_{\text{for a single PE}} \cdot 96 \cdot 0.25246 = frequency \cdot 12.118 \left[\frac{Ops.}{Cycle}\right]$$

Table 6.1 shows the *operations per second*, latency and resulting throughput for a batch size of 1 for several frequencies.

Table 6.1: Throughput, Performance, and Latency (for a batch size of 1) of the accelerator for the Baseline CNN Model.

	Frequency [MHz]			
	50	**100**	**200**	**250**
Ops. Per Second [GOPS]	0.60	1.21	2.42	3.03
Throughput [inferences/second]	61	123	245	307
Latency [msecs/inference]	16.30	8.15	4.08	3.26

6.1 Timing

The maximum frequency at which the accelerator can function without committing a slack violation is 200 MHz, being the critical path the one between the signal that carries the configuration parameter that indicates the height/width of the ofmap (*EF*) and the Data-In (*D1*) of the writing port of one of the double-port memories that compose the OFMAP memory Block. A considerable part of the delay is due to the routing interface that deals with all the memories that compose the OFMAP memory Block.

6.2 Area

The area is dependant of the technology used. The CMOS 28 nm SOI technology libraries provided by *STMicroelectronics* were used for this work. Route & Place was not implemented hence the net area is an approximate figure defined by the technology libraries.

The total area adds up to 1.1 mm^2, out of which 0.95 mm^2 corresponds to the cells and 0.15 mm^2 to the nets. Figure 6.1 shows a breakdown of the area occupation of the accelerator. Most of the area is employed in the SRAM memories. It is worth mentioning that even though the size of the Weight memory (80 KB) is bigger than the size of the OFMAP memory (64 KB), the latter occupies more than twice than the former. A reason for this is that this memory block is composed by eight separated embedded memories with its auxiliary logic, which increases the total area considerably. Furthermore, double port memories dedicate more area to auxiliary logic. Sequential elements occupy roughly 23% of the total area, and most of those are employed in the RFs of the PEs. The logic portion refers to multiplexers, basic gates such as *and* gates, and other multi-stage compound gates. The miscellaneous part refers to the inverters (0.73%), buffers (0.23%) and clock-gating related logic (0.03%).

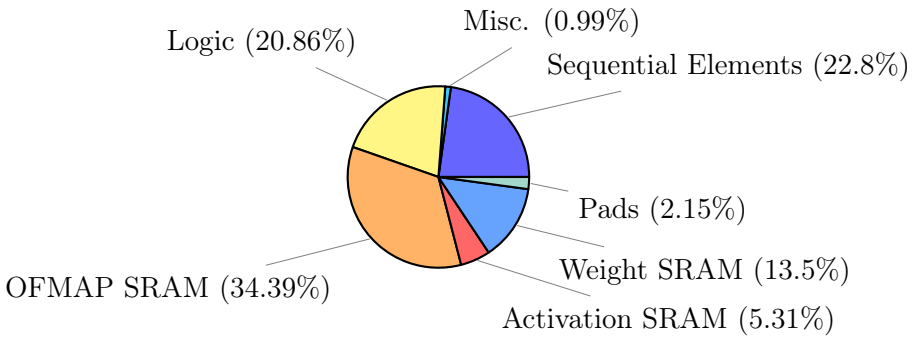


Figure 6.1: Area breakdown of the accelerator.

6.3 Power

The power consumption was estimated using switching activity with *Synopsys*' *PrimeTime* timing analysis tool. Clock-Gating is applied to the Register Files within each PE in the PE Array. This allows a tight control of the clock signal that feeds the Register Files. The clock enable gates the clock when the PEs are stalling and when the ofmap values are being stored back to the Activation memory. The number of clock cycles that the PEs are stalling plus the number of clock cycles in which the activations are stored back to SRAM accounts for 40% of the total number of the clock cycles that the architecture needs to compute the Baseline CNN Model. As a consequence, power consumption is reduced 40% as compared to when not applying Clock-Gating, achieving an overall power consumption of 50.01 mW. The energy efficiency of the architecture is 47.54 GOPS/W.

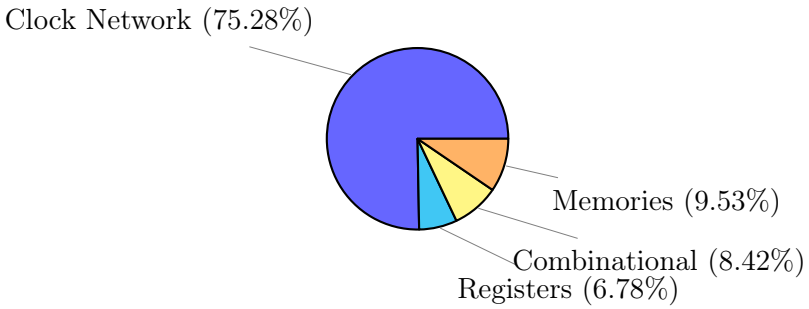


Figure 6.2: Power Consumption breakdown of the accelerator.

As it can be seen in Figure 6.2, most of the power budget is taken by the clock network (75.28%). The power budget is dominated by the clock network, since RFs within the PEs compose the 69% of the total number of registers in the system. As a comparison, in *Eyeriss V1* approximately 45% of the total power is consumed by the clock network [12].

6.4 Summary

Table 6.2 below summarises the obtained results in previous sections.

Table 6.2: Accelerator Specifications.

On-Chip SRAM [KB]	176
Number of PEs	96
Register Files Size (per PE) [B]	92
Storage Registers [KB]	8.77
Clock Frequency [MHz]	200
Network[1]	CNN
Data-width	Activations: 16-bit Weights: 8-bit
Dataset	CIFAR-10
Performance [GOPS]	2.42
Energy Efficiency [GOPS/W]	47.54
Natively Supported CNN Shapes[2]	Filter Size: 3x3 Channel Dimensions: 8x8, 16x16, 32x32 # of Filters: 1-256 # of Channels: 1-256 Stride: 1 Batch Size: 1

Table 6.3 shows a comparison of prior relevant work. Comparison with *Eyeriss v1* becomes relevant since much of the present work is based on Chen et al. contributions. At 200MHz, *Eyeriss v1* achieves an average performance of 23.1 GOPS. In the present work, even though all the PEs are utilized for computation, a single PE performs such computations only a 25% of the total processing time, which explains why this figure is significantly lower than in *Eyeriss v1*. On the other hand, *Eyeriss v1* achieves an energy efficiency of 83.1 GOPS/W, because it takes advantage of the sparsity of the network by avoiding ineffectual operations and by applying a compression algorithm to read/write from external memory.

Eyeriss v2 is a specialized architecture for accelerating compact versions of sparse CNNs, which are modified versions of a CNNs that improve computational efficiency by reducing the number of connections without affecting the accuracy. It upgrades the NoC from *Eyeriss v1* and incorporates a hierarchical mesh, that can adjust to the varying amounts of data reuse and bandwidth demands. This enhances the efficiency of the computation resources. It is evaluated using a sparse version of AlexNet, resulting in a performance of 153.6 GOPS and achieving an energy efficiency of 962.9 GOPS/W.

[1] Excluding FC layers.

[2] Channel size is limited by bitwidth of configuration parameters, but most likely constrained also by memory capacity.

VGG16 is a CNN architecture that was developed by the Visual Geometry Group (VGG) at the University of Oxford. It is composed of 16 layers, including 13 convolutional layers and 3 dense layers. It is widely used for evaluation and benchmarking. [33] uses a compact model of this CNN and apply dynamic quantization, reducing the bitwidth down to 16 bits for both weights and activations. *NullHop* exploits activation sparsity by applying zero-skipping without wasting any clock cycle, and implements a compression scheme optimized for sparsely activated CNN layers. When evaluating *NullHop* with VGG16, the accelerator's capability in efficiently computing highly sparse CNNs attains a performance of 420.83 GOPS and an energy efficiency of 2.7 TOPS/W.

Table 6.3: Comparison of the proposed architecture against other relevant works.

	Eyeriss v1 [12]	**Eyeriss v2** [15]	**NullHop** [33]	**This Work**
Technology	65nm	65nm	28nm	28nm
Area $[mm^2]$	4	-	8.1	1.1
On-chip SRAM [KB]	181.5	246	1,088	176
Model	AlexNet	sparse AlexNet	VGG16	-
Network	CNN			
Bit Precision *Activations/Weights*	16b/16b	8b/8b	16b/16b	16b/8b
Clock Frequency [MHz]	200	200	500	200
Performance [GOPS]	23.1	153.6	420.83	2.42
Energy Efficiency [GOPS/W]	83.1	962.9	2,715	47.54

Chapter 7

Conclusions & Future Work

This work introduces a CNN accelerator that adapts itself to the convolutional layer being processed and maximizes the PE utilization according to the size and shape of the layer. The accelerator implements a Network-on-Chip that allows a hierarchical memory scheme that minimizes data movement, pushing the input data as close to the computation units as possible. It does so by taking advantage of the Row-Stationary dataflow, which approach focuses on reutilizing the input data within the PEs for as much as possible before having to fetch new values from the on-chip SRAMs. As a consequence of this mapping strategy, memory utilization is optimized and energy consumption is minimized. In addition to this, using fixed-point quantization and Clock-Gating further improves the energy efficiency and area footprint of the accelerator. Lastly, the accelerator could easily be expanded to support kernel sizes other than 3, since the mapping strategy allows it.

Regarding possible improvements that are focused on reducing the memory overhead of the accelerator:

- **Dynamic Fixed-Point Quantization**. Dynamic fixed-point quantization [34] is a technique used to reduce the precision of the weights and activations, while maintaining accuracy. In dynamic fixed-point quantization, the precision is not fixed in advance, but is determined dynamically based on the dynamic range of the weights and activations. During training, the range of the weights and activations is monitored and a fixed-point format with the appropriate precision is chosen based on the observed range (usually on a layer-by-layer basis). This allows for more efficient use of the limited precision available in hardware, while still maintaining accuracy. For example, *Tensorflow* uses such technique by post-training the network, allowing 8-bit fixed-point values for both weights and activations with no accuracy drop. This improvement would reduce the area overhead of the memories dedicated for activations and psums, while also reducing the area of the computation units.
- Study the utilization of **pseudo dual-port SRAMs** for the OFMAP memory. This would reduce the area dedicated to the OFMAP memory considerably while keeping functionality and performance intact. A pseudo dual-port

SRAM uses a single port and an additional set of control signals to achieve the same functionality as a dual-port SRAM. The control signals allow the single port to be time-multiplexed between the two sets of data, allowing for simultaneous access.

- **Compression Techniques**. CNNs tend to have a high sparsity (i.e. a lot of zeros in their activation values), which increases as the layer is deeper. While reading the activations from an external DRAM into the on-chip SRAM, a compression algorithm could by implemented in the hardware as to avoid unnecessary reads/writes hence reducing the memory overhead. This requires the accelerator to work on a layer-by-layer processing basis, as opposed as how it works now that processes several layers sequentially in one run.

In addition, there are a countless amount of techniques that can also be implemented to the architecture that can further improve its performance on terms of energy efficiency, area, or throughput. A few of them are mentioned below.

- **Zero Skipping**. Again, by taking advantage of the sparsity property, when performing the MAC operations within the PEs, ineffectual computations, in which one of the operands is zero, could be avoided and hence improve the overall energy consumption.
- **Clock Gating**. Clock gating is a technique used to reduce power consumption. It involves selectively enabling or disabling the clock signal to certain parts of the circuit based on whether they need to be active or not. For example, when having a PE Array with five rows (Y = 5), if a convolutional layer with a kernel size of 3 is computed, the last two rows of PEs will not be used. By applying clock gating the overall energy consumption during the processing of this convolutional layer could be reduced.
- **Adding extra Features**. By extending the set of configuration parameters for each layer to be processed, it would be relatively easy to tailor the accelerator to the specific characteristics of the layer, and adding the extra hardware necessary for it. For example, applying average pooling, allowing computations of FC layers or allowing different strides other than one for the convolutional layers.
- **Reduce Stalling**. As it was described in Chapter 3, the PE Array must stall computation while waiting for the PISO Buffer to empty for $(E - R - S + 1) \cdot (F \cdot p \cdot (r - 1))$ clock cycles per pass, this time could be reduced and thus improve the performance of the accelerator. One way of achieving this is extending the word-length of the OFMAP memory four times the current width (from 32 bits to 128 bits), and modify the PISO Buffer to allow outputting four ofmaps at the same time and allocate them in the same memory address. This way the stalling time could be reduced x4 times.
- **Mapping**. The control logic of the accelerator could be improved for enhancing its mapping capabilities and allowing a wider set of natively supported CNN shapes. For example, by allowing allocation of multiple rows

of activations within the RF of the PE (i.e. allow parameter q to be higher than 1). Or, at the cost of more area and higher power consumption, improve the spatial mapping capabilities by increasing the *tilling* parameter (t), this would create instantiations of the PE Array and the Adder Tree, and would imply separate on-chip SRAMs for each set, but concurrent computing would be greatly improved. Another key mapping improvement would be increasing the scalability of the accelerator by allowing the convolutional network to be folded into the PE Array when the dimensions of the ifmap channel are wider than the size of the PE Array (e.g. allowing a conv. layer with ifmap dimensions of 64 x 64 to be computed in a PE Array with $X = 32$), thus improving the set of supported CNN shapes for a given accelerator configuration.

- **Design a mapper that can automatically search the best mapping option given network constraints**. Since it is possible to tailor the accelerator according to the hardware parameters prior to synthesise it, it would be interesting to analyze the set of CNNs that the accelerator is more likely to be working with depending on the application, and create a tool that, based on some constraints, could output a working design in a similar way a compiler would do.

References

[1] Teo Susnjak. "ChatGPT: The End of Online Exam Integrity?" In: *arXiv preprint arXiv:2212.09292* (2022).

[2] Avijit Ghosh and Genoveva Fossas. "Can There be Art Without an Artist?" In: *arXiv preprint arXiv:2209.07667* (2022).

[3] Fuxun Yu et al. "REIN the RobuTS: Robust DNN-based image recognition in autonomous driving systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2020), pp. 1258–1271.

[4] Sumit Saha. "A comprehensive guide to convolutional neural networks—the ELI5 way". In: *Towards data science* 15 (2018).

[5] Yin Chen et al. "Deep Learning for Healthcare: Review, Opportunities and Challenges". In: *IEEE Access* 6 (2020), pp. 37010–37028.

[6] O Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge (lLSVRC)". In: (2014).

[7] Shiguo Liu et al. "Deep learning in mobile and wireless networking: A survey". In: *IEEE Access* 6 (2018), pp. 37129–37160.

[8] Mi Zhang et al. "Deep learning in the era of edge computing: Challenges and opportunities". In: *Fog Computing: Theory and Practice* (2020), pp. 67–78.

[9] Georgios Flamis, Stavros Kalapothas, and Paris Kitsos. "Best practices for the deployment of edge inference: The conclusions to start designing". In: *Electronics* 10.16 (2021), p. 1912.

[10] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[11] Manoj Kumar Jain, M Balakrishnan, and Anshul Kumar. "ASIP design methodologies: survey and issues". In: *VLSI Design 2001. Fourteenth International Conference on VLSI Design*. IEEE. 2001, pp. 76–81.

[12] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138.

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Using dataflow to optimize energy efficiency of deep neural network accelerators". In: *IEEE Micro* 37.3 (2017), pp. 12–21.

[14] Xuan Yang et al. "DNN dataflow choice is overrated". In: *arXiv preprint arXiv:1809.04070* 6 (2018).

[15] Yu-Hsin Chen et al. "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[16] Zidong Du et al. "ShiDianNao: Shifting vision processing closer to the sensor". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 92–104.

[17] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[18] Andrej Karpathy et al. *A cartoon drawing of a biological neuron (left) and its mathematical model (right)*. [Online; accessed January 3, 2023]. 2022. URL: `https://cs231n.github.io/neural-networks-1/`.

[19] Vivienne Sze et al. "Efficient processing of deep neural networks". In: *Synthesis Lectures on Computer Architecture* 15.2 (2020), pp. 1–341.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[21] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[22] Chan Park, Sungkyung Park, and Chester Sungchung Park. "Roofline-model-based design space exploration for dataflow techniques of cnn accelerators". In: *IEEE Access* 8 (2020), pp. 172509–172523.

[23] Atul Rahman et al. “Design space exploration of FPGA accelerators for convolutional neural networks”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1147–1152.

[24] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. “Using dataflow to optimize energy efficiency of deep neural network accelerators”. In: *IEEE Micro* 37.3 (2017), pp. 12–21.

[25] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[26] Mingxing Tan and Quoc Le. “Efficientnet: Rethinking model scaling for convolutional neural networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[27] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[28] C Szegedy et al. “Going deeper with convolutions In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2015”. In: *Google Scholar* (2015), pp. 1–9.

[29] Boris T Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17.

[30] Suyog Gupta et al. “Deep learning with limited numerical precision”. In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746.

[31] Mantas Mikaitis. “Stochastic rounding: algorithms and hardware accelerator”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–6.

[32] Xucheng Ye et al. “Accelerating CNN training by pruning activation gradients”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*. Springer. 2020, pp. 322–338.

[33] Alessandro Aimar et al. “Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps”. In: *IEEE transactions on neural networks and learning systems* 30.3 (2018), pp. 644–656.

[34] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (2014).

Printed by Tryckeriet i E-huset, Lund 2023

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-917
http://www.eit.lth.se