POLITECNICO DI TORINO

College of Engineering

Department of Electronics and Telecommunications

M.Sc. Thesis

Design & optimization method of custom HW accelerators

Presented By:

Ahmed Bakry Hussein

Supervised By:

Prof. Andrea Calimera

October – 2020

DECLARATION

I hereby certify that this report, which I now submit for assessment on the program of study leading to the award of Master of Science in Electronic and Telecommunication Engineering, is all my own work and contains no Plagiarism. By submitting this report, I agree to the following terms:

> *Any text, diagrams or other material copied from other sources (including, but not limited to, books, journals, and the internet) have been clearly acknowledged and cited followed by the reference number used; either in the text or in a footnote/endnote. The details of the used references that are listed at the end of the report are confirming to the referencing style dictated by the thesis defense template and up to my knowledge, accurate and complete.*

I have read the sections on referencing and plagiarism in the thesis template. I understand that plagiarism can lead to a reduced or fail grade, in serious cases, for the Thesis defense.

Student Name: Ahmed Bakry Hussein

Student Number: S250789

Signed: _____

Date: – 10 – 2020

ACKNOWLEDGMENT

This accomplishment could have not been achievable without the endured support of the following people that I am so grateful for their efforts and I am so glad to present them my appreciation:

- **My Family:** for their continuous financial and moral support, which effected positively on the progress of this work.

- **My fiancée:** your continuous support and pushes during hard times were so heroic, I do really appreciate your stands with me.

- **Professor *Andrea Calimera*:** who gave me the keys of most problems and obstacles I faced in my way towards the completion the work. I want to appreciate him for his patience, effort and time elapsed in advising and guiding me.

- ***Antonio Cipolletta*:** I cannot thank him enough for the knowledge he gave me, time he spent on me, patience he had with him was extraordinary, and above all of that, his belief in me. I do really appreciate all your professional advices which effectively managed to end this work successfully.

ABSTRACT

The world is in hunger for computational abilities, day by day, the need for computing complex algorithms and using complex applications is increasing extensively, with such need, Electronics world is been brutally pushed against its limits, especially with the rise of using Artificial intelligence and its applications. However, these abilities not just require inventing new technology, but, also, require finding another and better way of processing heavy data using new dataflow, unlike the traditional one. Nevertheless, these new processing strategies must be able to process the data just like the normal way but might be with different dataflow. Since most of the complex applications nowadays require machine learning implemented on some neural networks, it is important to give an intense focus for neural network processing unit.

In the upcoming years, artificial intelligence will conquer our world with its tremendous applications which will be either be implemented on servers, micro-computers, supercomputers, etc. depending on the application. However, this opens a contradiction, how will it be possible to process these quite complex data on an embedded system? For instance, nowadays, most of machine learnings processing are being processed on graphical processor units, which requires relatively big area and energy hunger. So, the idea was to implement a different kind of processing unit, that requires less area, and much energy saver, with the same performance and in some scenarios, with better performance too. Also, instead of using some GPU that sometimes it is overqualified for some applications due to the huge parallelization processing capabilities, it is possible to dedicate a parametric capability, in which it could be just suitable for the required application. Not to mention that, the focus here for this kind of processing unit is to mainly solve the problem embedded systems world is facing with artificial intelligence. In other words, this kind of processing unit, must not always be the best processing unit for all neural networks, which means, it is possible to use a mixture of both the high parallelization technique used in GPUs with this kind of processing, but as this is not always feasible for all applications due to area and power constraints, this processing unit could be the best option for processing neural networks so far.

As said before, the difference of processing data and its flow makes it suitable for machine learning applications, but for other applications in signal processing like videos processing and gaming applications, GPUs will give better results since both of these applications hold big set of frames that needs to be processed all at once using for instance the huge number of ALUs inside the GPU chip.

TABLE OF CONTENTS

List of figures **8**

list of tables **11**

List of Equations **12**

1 Introduction **13**

1.1 The aim of this work 17

2 Related works **18**

2.1 Neural networks 18

2.1.1 Neural network history 18

2.1.2 Neural network background 20

2.2 Neural network accelerators 26

2.2.1 Hardware architectures 26

2.2.1.1 Temporal architecture 26

2.2.1.2 Spatial architecture 27

2.2.1.2.1 Systolic Array 28

2.2.1.2.1.1 Input stationary 28

2.2.1.2.1.2 Weight stationary 30

2.2.1.2.1.3 Output stationary 31

2.2.1.3 Real life AI accelerator examples 31

2.2.1.3.1 TPU 31

2.2.1.3.2 Eyeriss 33

2.2.1.3.3 nvidia ga100....34
2.2.1.3.4 intel stratix 10 fpga....36
3 Neural network accelerator design....38
3.1 Chisel....38
3.1.1 Scala build tool....39
3.1.2 Flexible Internal Representation (FIR) for (RTL)....40
3.2 Verilator toolchain....40
3.3 Design Anatomy....41
3.4 convolution layer anatomy....41
3.4.1 Processing unit design....45
3.4.1.1 Processing array....50
3.4.1.1.1 Processing element....51
3.4.1.1.2 Internal registers....55
3.4.2 Partial products accumulation....56
3.5 System design....60
3.5.1 Scala code....60
3.6 Work contribution....72
4 Experiments and results....73
4.1 Synthesis and optimization....73
4.1.1 Design synthesis....73
4.1.1.1 Design compiler....74
4.1.1.1.1 Design-ware....74

4.1.1.2 Prime time shell 74

4.1.2 Area analysis....75

4.1.3 Power analysis 77

4.2 Latency analysis....77

4.3 Memory accesses analysis 80

4.4 Data analysis....82

5 Conclusion 83

5.1 Future Work....84

References 85

LIST OF FIGURES

Figure 1-1 Supervised ML, [4] **14**

Figure 1-2 Unsupervised ML, [4] **14**

Figure 1-3 Reinforcement ML, [6] **15**

Figure 2-1 Artificial intelligence subfields, [12] **18**

Figure 2-2 History of Neural networks, [15] **19**

Figure 2-3 Structure difference between the biological and the artificial neuron, [18] **21**

Figure 2-4 Synapse detailing [20] **21**

Figure 2-5 Types of activation functions **22**

Figure 2-6 Simple neural network example, [21] **22**

Figure 2-7 Neural network architecture zoo, [22] **23**

Figure 2-8 Recurrent Neural Network, [23] **24**

Figure 2-9 Convolution layer calculation **25**

Figure 2-10 Temporal Architecture (SIMD/SIMT), [24] **27**

Figure 2-11 Data fetching energy cost, [25] **27**

Figure 2-12 Data stationarity bandwidth comparison, [26] **28**

Figure 2-13 Matrix Multiplication using input stationary dataflow **29**

Figure 2-14 Matrix Multiplication using Weight stationary dataflow **30**

Figure 2-15 Matrix Multiplication using Output stationary dataflow **32**

Figure 2-16 TPU Architecture **33**

Figure 2-17 Eyeriss dataflow used, [24] **34**

Figure 2-18 Eyeriss Architecture, [29] **34**

Figure 2-19 Nvidia GA100 Top-level Architecture, [30] **35**

Figure 2-20 Nvidia GA100 Tensor core, [31] **35**

Figure 2-21 Nvidia GA100 Matrix Sparsity, [30] **36**

Figure 2-22 Intel Stratix 10 AI Tensor blocks, [32] **36**

Figure 2-23 Neural network pipelining, [33] **37**

Figure 3-1 Convolution layer parameters visualization, [34] **43**

Figure 3-2 *GeMM im2col* strategy, [35] **44**

Figure 3-3 Top view of the *Processing Unit* **45**

Figure 3-4 Organization cycle sample **47**

Figure 3-5 Input shifting cycle sample **48**

Figure 3-6 Processing Unit synchronization waveform **49**

Figure 3-7 Processing Array top view **50**

Figure 3-8 Matrix multiplication sample **51**

Figure 3-9 Processing Element Top view **52**

Figure 3-10 Processing Element top view of the first row of the Processing Array **52**

Figure 3-11 Sample of processing a Weight (4x4) with Input tensor (4x2) in Processing Array with hardware components **54**

Figure 3-12 Sample of accumulating Weight (8x8) with Input tensor (8x8) **57**

Figure 3-13 Completion of processing and accumulation a Weight (4x4) with Input tensor (4x4) in Processing Array with hardware components **59**

Figure 3-14 Neural network parameters definitions flowchart **60**

Figure 3-15 Kernel distribution flowchart **61**

Figure 3-16 Input tensor distribution tensor flowchart **62**

Figure 3-17 Processing aspects declaration flowchart **63**

Figure 3-18 General cycle flowchart **65**

Figure 3-19 Data placement stage flowchart **66**

Figure 3-20 Data placement stage by row flowchart **67**

Figure 3-21 Data placement stage by column flowchart **68**

Figure 3-22 Data processing stage flowchart **69**

Figure 3-23 Data processing stage of the upper part **70**

Figure 3-24 Data processing stage of the lower part **71**

Figure 4-1 Sequential vs Combinational cells area analysis in all configuration 76

Figure 4-2 Normalized latency with respect to 8x8 configuration on the proposed benchmarks 79

Figure 4-3 Latencies in benchmarks having 115.6 * 10^6 memory accesses..... 81

Figure 4-4 Latencies in benchmarks having 57.8 * 10^6 memory accesses 82

LIST OF TABLES

Table 2-1 comparison between Biological neural system and Modern computers, [17] 20

Table 4-1 Area analysis of multiple configurations of Processing Array 75

Table 4-2 Power analysis of the Processing Array in multiple configuration 77

Table 4-3 10 different benchmarks tested on the proposed configuration 78

Table 4-4 Benchmarks having 115.6 * 10^6 memory accesses comparison 81

Table 4-5 benchmarks having 57.8 * 10^6 memory accesses 81

Table 4-6 Area, Power, and Maximum, Average, and minimum latencies comparison in different benchmarks 82

LIST OF EQUATIONS

Equation 2-1 Rosenblatt perception mathematical approach 22

Equation 2-2 2x2 Difference between TA and SA memory accesses 29

Equation 3-1 Number of iterations calculation in the general cycle 46

Equation 3-2 Number of iterations calculation in the organization cycle 47

Equation 3-3 Number of iterations calculation in input shifting cycle 48

Equation 3-4 Starting condition of Partial products shifting cycle 48

Equation 3-5 Number of iterations calculation in Partial products shifting cycle 49

Equation 3-6 Total number of iterations calculation 49

Equation 3-7 Number of Input buffer calculation 55

Equation 3-8 Number of Processing element buffer 55

Equation 3-9 Number of Input shifters buffer 55

Equation 3-10 Number of Weight shifters buffer 56

Equation 3-11 Number of Output shifters buffer 56

Equation 4-1 Number of sequential cells estimation 76

Equation 4-2 Number of combinational cells estimation 76

Equation 4-3 Latency calculation in milliseconds 79

Equation 4-4 Memory accesses calculation 82

Chapter 1

1 INTRODUCTION

Since the rise of the industrial revolution, the need for controlling machines and digital world is intensively increasing, with this revolution a rise in computer science has gained a great fame to serve lots of aspects when it comes to control and automation.

No wonder why machine learning is taking a huge part of our daily life, it has been effectively upgrading the life form of humanity. Moreover, it is possible to predict events, it would not have been possible without it, throughout the algorithms and mathematical approaches it is using. One example to explain how powerful machine learning is, in a study made by *Joseph Risi* and other scientists, [1], they studied around two million declassified electronic cables used for communication between the United States of America and its embassies between 1973 and 1979. However, these cables were not classified according to their importance, like for instance the social events and the events that require much more attention. Nevertheless, when comparing the important cables with some other important ones using machine learning model, around less than 1% were already recognized by historians. Furthermore, this concludes that, even though it is very hard for historians to analyze the huge amount of historic data accordingly to their importance, yet, using machine learning it has become more feasible.

It is certain that, it is shaping out the future, and day by day, its applications are effectively increasing, [2]. Nowadays, it has been widely used in applications like: Image processing, Language processing and translation, Route detection, Speech recognition and Forecasting.

Machine learning is a technique used for data analyzing, that basically from having lots of data, it explains to digital computers how to react, more likely as naturally happens to humans and animals. Moreover, the type of algorithms that are used in ML are computational methods, which learns data on its own. However, there are three types of training in machine learning, [3]:

1. Supervised learning
2. Unsupervised learning

3. Reinforcement learning

Depending on the data it is harvesting, the type of machine learning is used. In general, the difference between the first two types, Supervised and Unsupervised learning is that in the Supervised learning the algorithms used will train on labeled data, which means, the data that will be used, are classified into labels, taking Figure 1-1 as example, given four types of cats, that were trained out by an AI model, and these cats were labeled as “Cats”, after that, when using this model to predict out cats among other animals, which in this case, 2 dogs and 2 cats, the prediction came out as expected, it recognized cats among the animals with its label “Cats”, while the unknown animals, which haven’t been trained out before, were all recognized together as “Not cats”, which refers to unclassified animal.

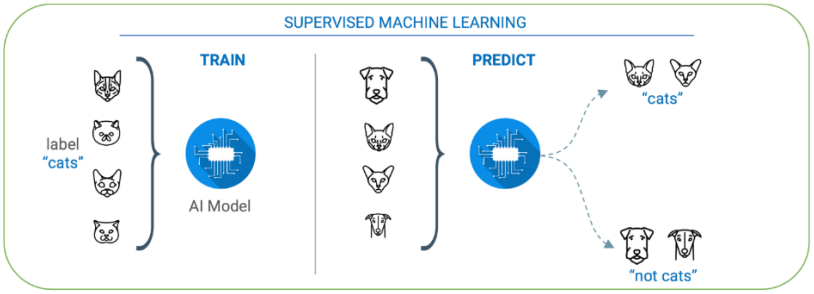


Figure 1-1 Supervised ML, [4]

Unlike the supervised ML, the algorithms it is using are training on unlabeled data, taking Figure 1-2 as example:

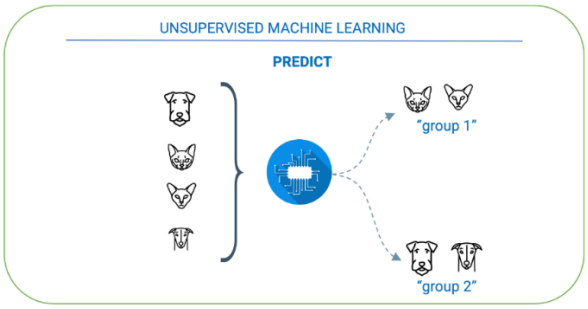


Figure 1-2 Unsupervised ML, [4]

The training dataset used in the Supervised version is relatively smaller than the one in the Unsupervised one, since the Unsupervised has the capability to work on its own, and find difference on itself, which means it doesn't require humans, so large set of training data is required to identify the difference. There is also, a Semi-Supervised type, whereas set of data have labels to help out training to predict the unlabeled one. In other words, it is a mixture of both, labeled and unlabeled data, [5].

The last type is the Reinforcement learning, which basically learns out from outputs, so it is basically a trial-and-error technique. Take the example in Figure 1-3 to understand Reinforcement learning.

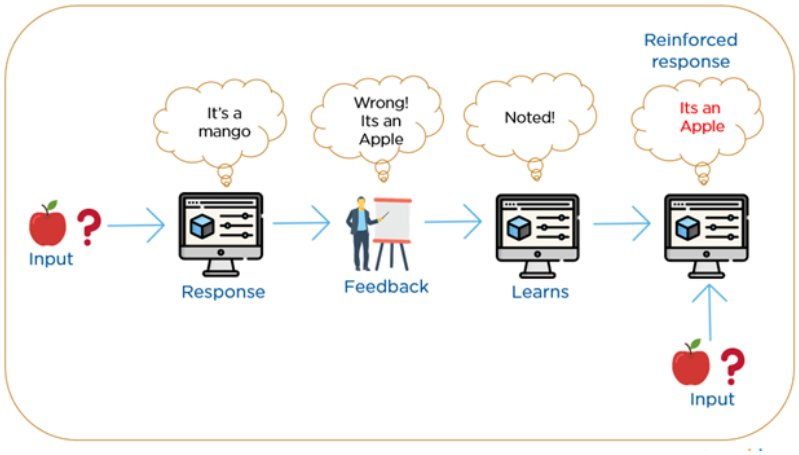


Figure 1-3 Reinforcement ML, [6]

Neural network ideology was based on the way humans' brains can solve problems, just like humans' biological neurons that transmits data allover, creating information learned or even interrupting for an event, etc. Humans brains are composed of a very complex structure with a very complex network connecting those biological neurons together, [7].

To sum up, neural network is a technique a computer can use to learn out from data, these data is affected by a collection of neurons to give out an output eventually, [8].

As discussed previously, as this network can learn out from data and process it, it has the ability to solve out these complex problems and keep training unstoppable. By time, the relying on it is

heavenly increasing, since it learns from the faults and the more it faces problems, the more it learns and adapt, [9].

Now, after understanding the capability of processing such complex data using neural networks, it will open up a portal to a serious problem, Electronics world is facing nowadays, which is, how may we build an accelerator capable to accelerate this kind of network, especially on a low-scale architecture, which has the power withstand its complexity with relatively smaller amount of cells.

No wonder that, some neural network applications require high degree of complexity, which require lots of Arithmetic logic units to execute them, but, in the same time, these applications require low-scale architecture as well, making it quite hard to implement these applications, [10], or be forced to use other solutions that may cost a lot. Neither the memory required will accommodate its complexity.

In this work, the focus is on accelerating the neural network from hardware perspective much more than optimizing its data. However, as the conventional way nowadays using large-scale architecture capable to withstand the complexity of neural networks, the solution was either to find a way to optimize this large-scale by finding defects and optimize the critical paths or change entirely the way of accelerating neural networks. In addition to that, FPGA is showing a very promising performance, due to its low power consumption, reconfigurability, and real-time processing capability, [11]. Another way is finding a solution to minimize the number of memory accesses from and to the main memory, besides, the possibility of flowing data all along between ALUs, something that can be close to the way neurons in the neural network connected together.

The solution presented in this work, is creating a framework having a new highly optimized High-level HDL called *Chisel*, that is designed to use a strategy of flowing data non-stoppable along relatively smaller amounts of ALUs. This strategy is depending on an architecture called spatial architecture, which gives the possibility to transfer data from and to ALUs, without the need to rely on the main memory in each clock cycle, like in temporal architecture. Moreover, the kind of flow along the processing elements used here is called *Systolic Array*, which gives the possibility to maximally save the usage of certain data, by keep reusing them internally, without the need to

return back to the main memory. However, this flow has three types, the one that is used here is called *Weight stationary.*

The difference in this solution that the processing element here is composed of a register-file and an ALU, not just an ALU used in CPUs and GPUs, for instance. It is possible to use the power of parallelization technique, just like GPUs, but with a limited number of memory accesses, and certainly with better energy efficiency, which makes this solution in general speaking, better than GPUs, except in some cases. Above that, in this work, the design is parametric, in other words, it is possible to just configure the number of the processing elements. Furthermore, increasing the number of the processing elements here, does not always increase the performance, in fact, overly increasing the processing elements, will lead to worsen the performance and increase energy consumption, so a tuning is required.

In addition to the HDL, a programming language called *Scala* was used to verify the HDL, convert the neural network into matrix multiplication and simulate the HDL.

1.1 THE AIM OF THIS WORK

This work aimed to assess the performances of different configurations of *Systolic array* using *Weight stationary* strategy on common neural networks workloads. Nevertheless, to also demonstrate the importance of having flexibility in the early phase of the hardware design process.

In order to guide this book, chapters were organized to the following:

- Chapter 2: Describing the explaining the terms and all the related works that has a connection with the work done here, in addition to giving the up-to-date examples of implemented works done by others.

- Chapter 3: Deeply explaining the work done here and showing how beneficial this work presenting.

- Chapter 4: Showing the results after testing out this accelerator using some benchmarks and analyzing all the resulted values.

Chapter two

2 RELATED WORKS

2.1 NEURAL NETWORKS

Neural networks has been given a huge focus in computer science, due to its abilities to compute complex algorithms. However, it is a subset from Artificial intelligence field as shown in Figure 2-1.

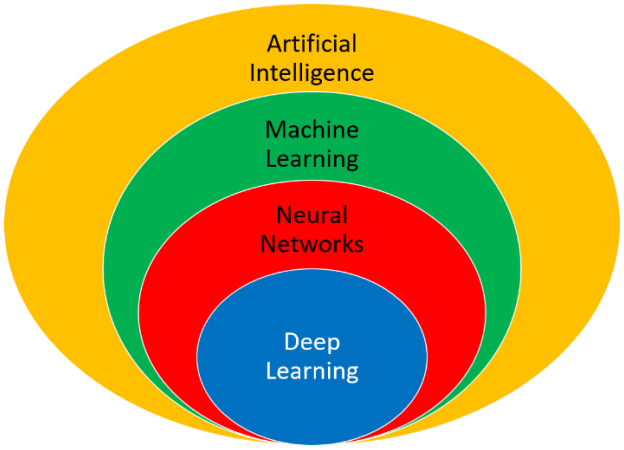


Figure 2-1 Artificial intelligence subfields, [12]

Neural network can be described as a huge network, that contain a set of algorithms that manipulate relations between inputs and outputs, [13], these algorithms used for the learning process.

2.1.1 NEURAL NETWORK HISTORY

The work of neural networks launched back in the 1940's, where scientist tried to build up models of the brain; it has been one of the newest and oldest scientific research interests. However, it all started back in the 1950's, when scientists implemented perceptron, and this will be explained in the upcoming sub-chapter, [Neural network background], it is basically a simple precursor of linear models. Initially, it solved quite simple problems, that was thoughtful to bring up the hope to solver more complex ones. However, when scientists tried out, it showed very flawed work, due to the limitations of the perceptron back then; it was not even possible to solver an XOR problem, so

there was no big benefit from that, until *Marvin Minsky* and *Seymour Papert* claimed in their book that, perceptron is very limited, [14].

In the 1980's, the work in neural networks has resurrected; scientists found a way to connect multiple perceptrons, this gave a huge boost for the computational ability, and possibility to solve more complex problems, back then in the 1950's was not possible to solve. In addition to that, during the same period, scientists were able to put up weights, where it gives the possibility of learning out and training from data, using the newly developed at that period, which is back propagation. Moreover, back then, it was still not so popular and data it had to train on were very limited, besides, number of people who were able to implement these kinds of networks were so limited. Not to mention the poor computational hardware capabilities that did not widely support the requirements that neural networks back then required. Furthermore, when hardware capabilities have increased and gave a chance for neural networks to be computed on, the motivation for it has increased, and programmers started to figure out how to train these networks in a large scale, allowing it to be used progressively in many fields.

In Figure 2-2, shows a brief history about neural network starting from year 1943 to 2019.

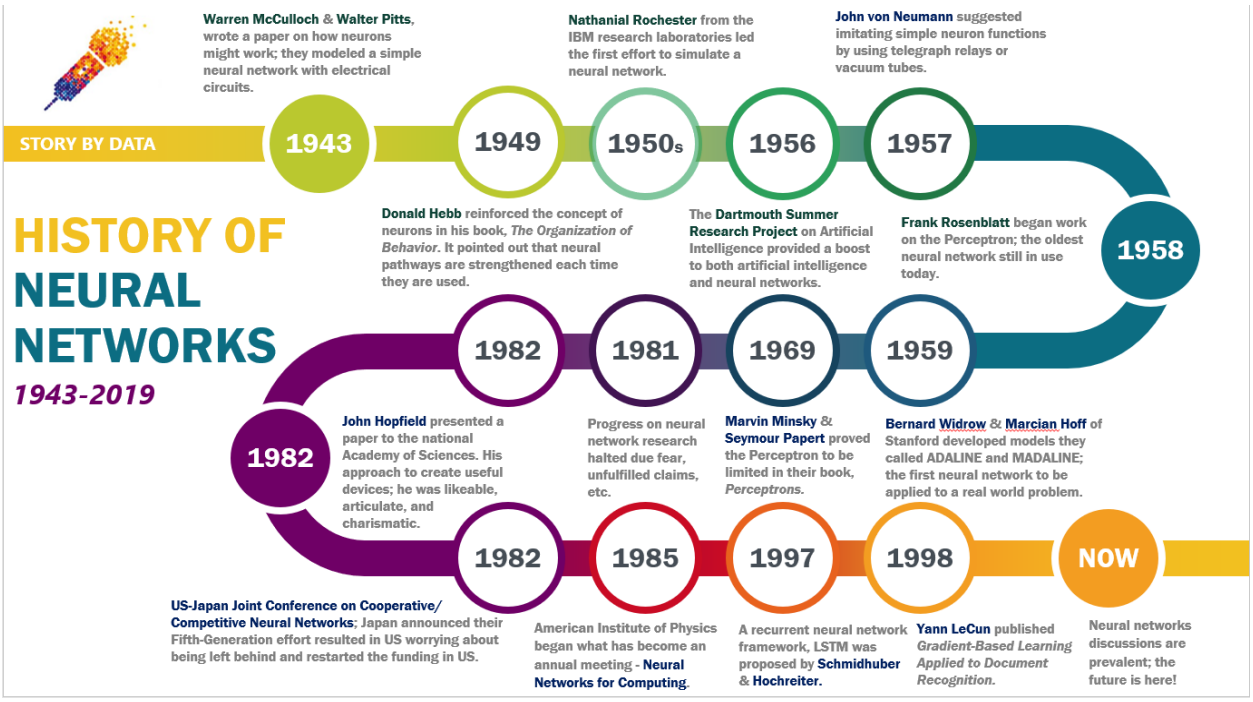


Figure 2-2 History of Neural networks, [15]

2.1.2 NEURAL NETWORK BACKGROUND

As introduced in the beginning, the neural network is considered to be a huge interconnected network of neurons that contains set of connections that effect on the set of inputs, to produce an output. However, these connections contain configurable parameters that are affected by the process of back propagation, throughout learning processes. This ideology is brought back to the way, how humans think, this is why, sometimes, when referring to neural network neurons, it is called artificial neurons, to differentiate between it and the biological neurons. Moreover, neural network has the power to process and compute environments relatively efficient comparing to humans or animals, [16]. With such powers, it has the possibility to recognize foreign languages, differentiate between humans sounds, fault detections in some computer systems, etc. By comparing biological neural system and modern computers, it is possible to find out from Table 2-1 that:

Biological neural system	Modern computers
Large amount of slow processing elements	Small amount of fast processing elements
Having internal integrated memories within the processing elements	Having an external segregated fast memory
Processing is done throughout self-learning	Processing is done throughout programmed environments
It is very powerful and can fix issues by itself	It can have errors and faults
The operations are not classified and has no boundaries	The operations are classified but has boundaries

Table 2-1 comparison between Biological neural system and Modern computers, [17]

As the biological neuron is a huge topic to discuss and it is not the aim of this work, it will be summarized in few lines to recognize the structure difference, between it and the artificial one, how was the artificial neural structure idea came from.

Biological neural is composed of three main parts:

- Dendrite: It is basically a wire-like, that is connected to the surrounding neurons to receive signals from them as an input to the Soma

- Soma: It contains the first anatomy part of the neuron, which has the nucleus where data is processed inside. In other words, it can be described as the core of the neuron.

- Axon: It is like a branch, where it receives the signal from the Soma, and transfer it to the Axon terminal, where it is connected to other neurons throughout Synapses. However, Synapse is the neural signal, where it is usually either a chemical diffusion or electrical impulse.

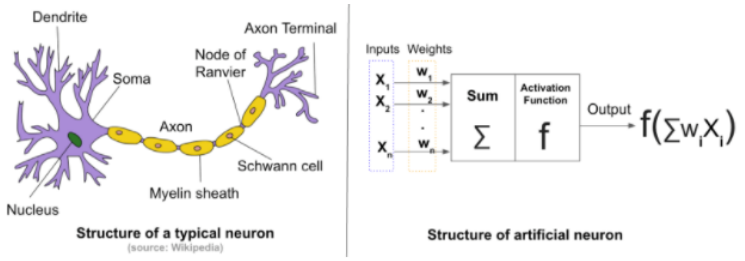


Figure 2-3 Structure difference between the biological and the artificial neuron, [18]

Depending on the signal [synapse] received from the dendrite, the neuron is either activated or deactivated, after neural summation process is done. In Figure 2-3, the difference between both of the neurons is showed. However, the structure of the artificial neuron is quite similar, the perceptron showed in this figure is called Rosenblatt perceptron, [19], for further understanding of the Synapse, refer to Figure 2-4.

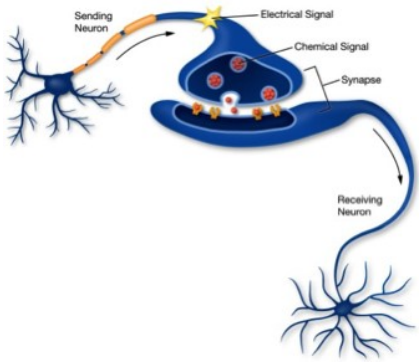


Figure 2-4 Synapse detailing [20]

Rosenblatt perceptron is a modulization for an artificial single neuron, that sums up a collection of weighted inputs, gained during training process. However, if the summation is greater than a specific value (Threshold), that is specified in the activation function part, the neurons produces, either 1 or 0, this can be understood from Equation 2-1.

$$output = \begin{cases} 1 & \sum_{n=1}^{\infty} W_n * X_n \geq \theta \\ & if \\ 0 & \sum_{n=1}^{\infty} W_n * X_n < \theta \end{cases} \qquad \text{Where } \theta \text{ is the threshold}$$

Equation 2-1 Rosenblatt perception mathematical approach

Usually, a generalization of the Rosenblatt perceptron is based on substituting to the step function, while there are other activation functions, such as Step, Linear, Sigmoid, Tanh and Relu. In Figure 2-5 the graphs for these functions is showed.

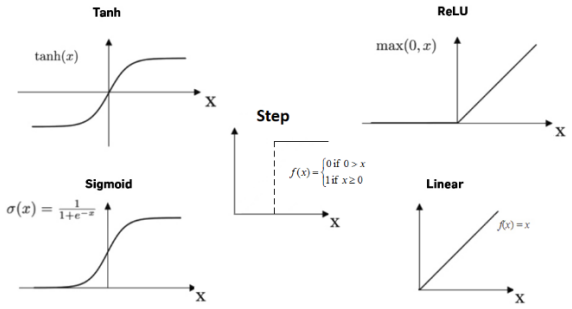


Figure 2-5 Types of activation functions

After explaining what the single neuron is, what does it compose of, it is possible now to discuss, what if this neuron implemented multiple times, making a network. Firstly, in general, a simple neural network is composed of three layers which are Input layer, Hidden layer, and Output layer. However, a simple network is shown in Figure 2-6 and how they are interconnected.

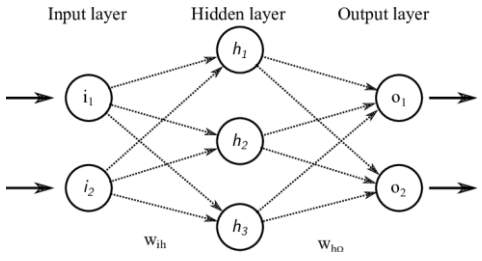


Figure 2-6 Simple neural network example, [21]

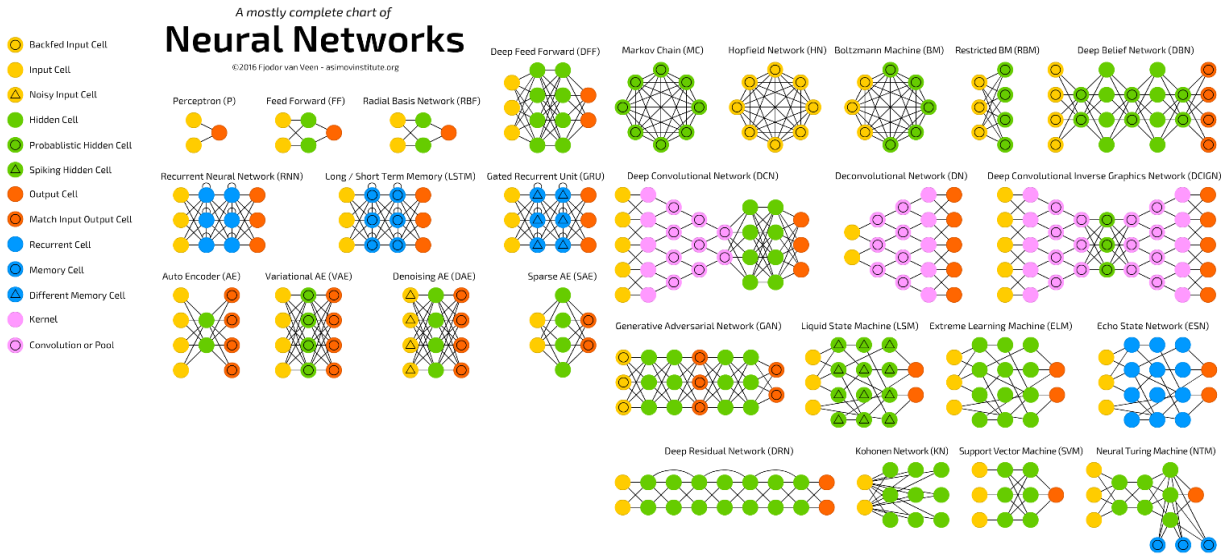


Figure 2-7 Neural network architecture zoo, [22]

The input layer is holding the set of data that is injected either by the user, or sometimes, a feedback from the output layer. However, this input layer passes by single or multiple hidden layers, which are affected by weights that effectively change out the input data. Moreover, complexity increases by increasing those hidden layers. Nevertheless, depending on the connections between these layers an architecture of neural network is formed; in Figure 2-7 a representation for almost all architectures of neural networks is shown.

The most commonly used architectures are:

- Feed Forward Network (FFN): This is the most popular one, its idea is straightforward, each neuron in each layer is connected to the other neuron in the layer after it. Each layer

Advantages:

1. It can be used for almost anything and easily to be used.
2. Solving problems is guaranteed.

Disadvantages:

1. Finding the best answer is not always capable, since it guarantees finding the local minimum not the global one.
2. The network parameters must be specified in the very beginning.
3. Complexity increases by increasing the input data.

learns relatively simple information and passes it to the layer after it, so it guarantees giving a correct answer but not always the best one, when given enough resources.

- Convolution Neural Network (CNN): This architecture is most popularly used for analyzing images, since it has the ability to detect patterns. For instance, patterns detection in images is very essential, since it can detect curves, edges, color similarities, shapes, etc. in a single image through-out filters, and this what makes it very useful when using any kind of image processing. Basically, it is used because it can share weights among neurons. Moreover, its process is divided into feature extraction and then like in FFN, it is data training.

  Advantages:

  1. Features extraction will increase by increasing layers. Hence, better data analyzing.
  2. Relatively fewer resources when comparing to FNN

  Disadvantages:

  1. Data hunger.
  2. Filters dimension needs to be as minimum as possible, so it does not overlap or skip an image parameter.
  3. The network parameters must be specified in the very beginning, yet, not as much as FNN.

- Recurrent Neural Network (RNN): This architecture is used when interpreting sequential information, like autocorrection. RNNs use other data points in a sequence to make better predictions. As the previous types, it has inputs and output, but what is different is, it is reusing data activation from previous nodes to affect the output as shown in Figure 2-8.

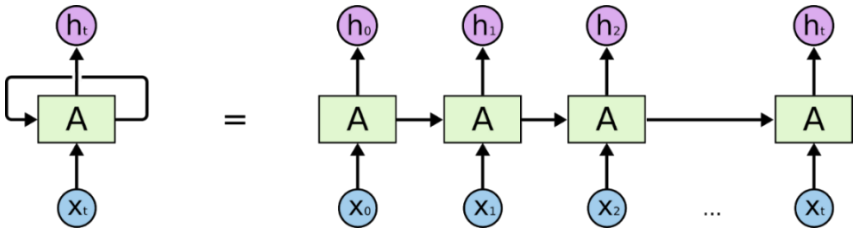


Figure 2-8 Recurrent Neural Network, [23]

Advantages:

1. Very effective with sequential data.
2. Function learning is very effective and keep back propagating it.

Disadvantages:

1. Because of reusing data, old data are easily lost.
2. Very sensitive to the data initiated with

These NNs cannot be understood out by any HW, because as it appears, it is multi-dimensional, and the input data will be affected out by weights and activation function to produce an output. Yet, the way the input data are affected is not very understandable by HWs too, taking for example the convolution layer architecture, which has the most focus in this work, the input will be multiplied with many weights, for many times, as shown in Figure 2-9, and this exploits the key for data reusing possibility, which will deeply be explained in the next chapters.

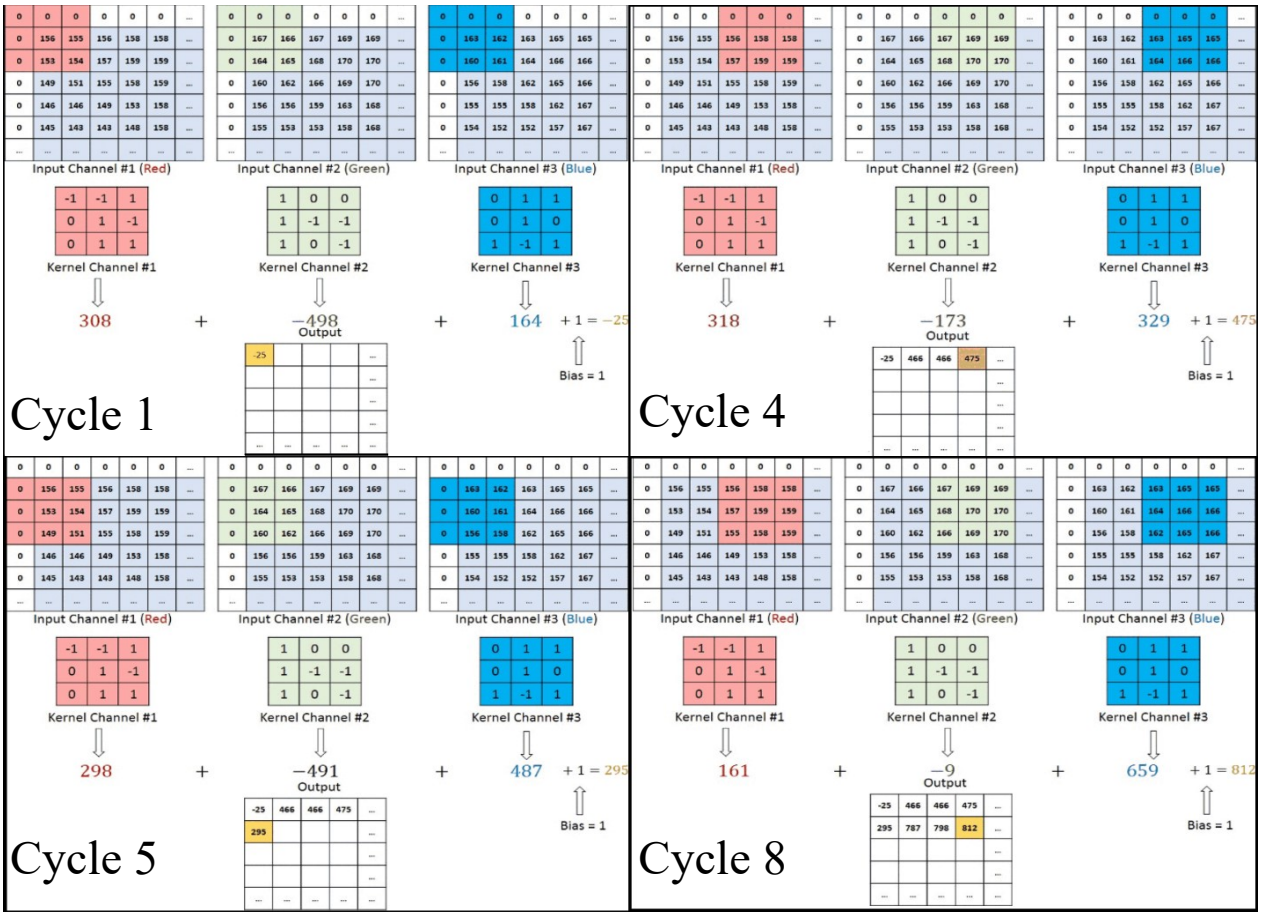


Figure 2-9 Convolution layer calculation

2.2 NEURAL NETWORK ACCELERATORS

After understanding the complexity of computing a neural network and how important did the development of computer architectures developed and improved the usage of using neural network upon history.

In this chapter, some architectures that is used in this work will be deeply explained, in addition to explaining some other similar architectures that could be used in future work, besides, giving some up-to-date accelerators that are developed by others.

2.2.1 HARDWARE ARCHITECTURES

Nowadays, most neural network accelerators use parallel computation paradigm for achieving an adequate performance, in contrast, this effects out lots of aspects in return. In brief, processors architectures are divided into two architectures, which are temporal architecture and spatial architecture that is used in this work. Generally speaking, most CPUs (Central Processing Units) and GPUs (Graphical Processing Units) use temporal architecture, where it has highly number of ALUs (Arithmetic Logic Units), which in this case the mostly used component is MAC (Multiply and Accumulator) that are all connected to a fast memory, but they cannot communicate directly with each other's. On the other hand, the spatial architecture is used in ASICs (Application Specific Integrated Circuits) and FPGAs (Field-Programmable Gate Arrays), where it can have limited number of ALUs from the perspective that overly increasing the ALUs can significantly severe the performance, power and area, but they can directly communicate with each other's. In the following chapters, the explanation of both architectures will be discussed.

2.2.1.1 TEMPORAL ARCHITECTURE

In multiprocessors architectures, each processor always has its own cache memory, where it has its own data path along all the components it has. However, when mapping a neural network on this architecture, a problem is faced, even though the performance could be quite high, it is not very energy efficient, in a sense that, ALUs only fetches out data from memory and cannot reuse data, so there are a lot of memory accesses, basically, they all share the same memory and control unit. Another thing, to improve the parallelism, there are techniques that can be used, like SIMD

(Single Instruction, Multiple Data) or SIMT (Single Instruction, Multiple Threads). In Figure 2-10 a generic visualization is shown.

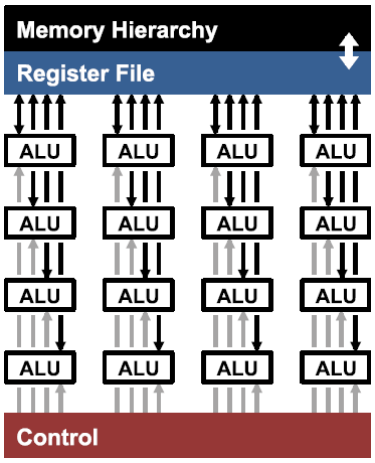


Figure 2-10 Temporal Architecture (SIMD/SIMT), [24]

2.2.1.2 SPATIAL ARCHITECTURE

The spatial architecture severely reduces the memory accesses from and to the DRAM (Dynamic Random-Access Memory). In the normal scenario, when using the ALU, at least three memory accesses has to be established, since three parameters are required, kernel weight, input tensor and partial sum which comes from previous ALUs. However, the idea here is to extend the usage of any of these parameters, by fixing them inside a RF (Register File) that is integrated inside the ALU, usually this unit is called PE (Processing Engine). By doing so, and also bypassing them towards the neighboring PE, data reusing can be accomplished, so, memory accesses will be reduced, in a sense that, there is no need to read one of the parameters each time, hence, better energy efficiency. Usually fetching out data from DRAM is more energy costly 200 times than from RF.

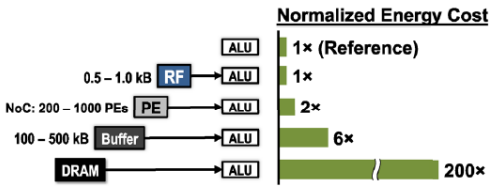


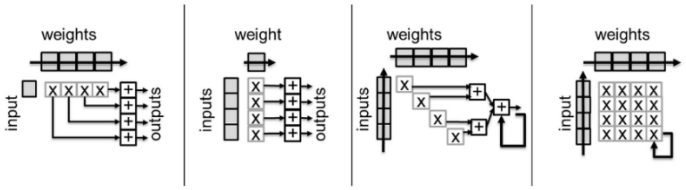
Figure 2-11 Data fetching energy cost, [25]

There are a lot of companies that gave an intense focus on implementing this idea, most famous one is Google TPU that is based on an architecture called systolic array.

2.2.1.2.1 SYSTOLIC ARRAY

Now after the possibility of connecting out all the PE, one possible implementation of a spatial architecture is the systolic array. Systolic name came out Systole, which is the idea of pumping out blood from the heart in a medical terminology. However, the idea is very similar, while saving out data, which are stationary, it is possible after using the non-stationed data, to pass them along the PEs next to them in the same row and use them with the other stationed data.

The data that can be stationed here can be one of the three parameters that are required. Depending on the shape of the neural network, the most energy efficient dataflow type can be used. In Figure 2-12, the difference in terms of bandwidth and PE implementation for different stationary dataflow.



	Input-stationary *(weight parallel)*	Weight-stationary *(input parallel)*	Output-stationary	Hybrids
Input BW	low	high	high	medium
Weight BW	high	low	high	medium
Output BW	high	high	low	medium

Figure 2-12 Data stationarity bandwidth comparison, [26]

2.2.1.2.1.1 INPUT STATIONARY

In this type, the input tensor data will be saved inside the register file of the PE. Usually both data, input tensor and Kernel weights are received from 2 separate buffers, and these buffers provide the PE. In order to understand the dataflow of this type, a visualization is shown in Figure 2-13.

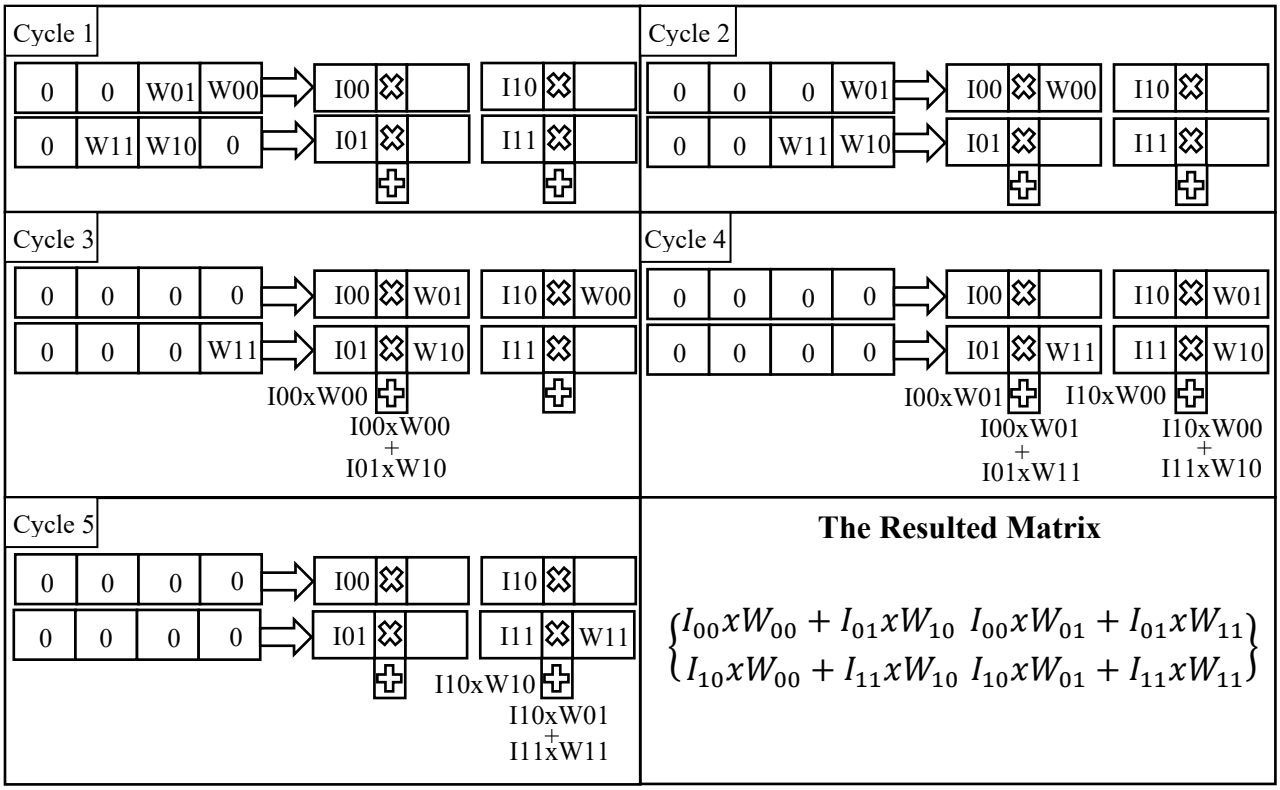


Figure 2-13 Matrix Multiplication using input stationary dataflow

Taken the example above, the number of DRAM memory accesses required here were 8 times, which basically, one access per each data. In the conventional temporal architecture, the number of DRAM memory accesses that will be required are 24, which basically each time an operation occurred including the accumulation of partial products. This gives superior saving in terms of energy consumption, by referring to Figure 2-11, it is possible to say that TA (Temporal architecture) will cost more by:

$$200 * (TA\ DRAM\ Accesses - SA\ DRAM\ Accesses) - 6 * SA\ Buffer\ Accesses - RF\ Accesses = 200 * (24 - 8) - 6 * 8 - 8 = 3144$$

Equation 2-2 2x2 Difference between TA and SA memory accesses

Even though that are different memory accesses occurs in SA, yet the most effective one is the DRAM one, hence, it generated this difference between both architectures, the SA is less around 34.5%. Certainly, this is not always the case, in fact, this is a very special one, yet, from a general

perspective, TA will always use much more DRAM accesses, hence, SA will always be more energy efficient.

2.2.1.2.1.2 WEIGHT STATIONARY

This type is very similar to the input stationary one, instead of stationing inputs inside the PE, weights will be saved. However, a similar visualization is shown in Figure 2-14 to understand its dataflow.

As shown, it is quite similar to input stationary, the key behind using this type is always the shape of the neural network. As explained before, what is a neural network constructed from and in convolution layer anatomy sub-chapter, the explanation of the parameters of a convolution network, and how it is converted into a matrix multiplication, however, depending on these parameters the type of systolic movement is chosen.

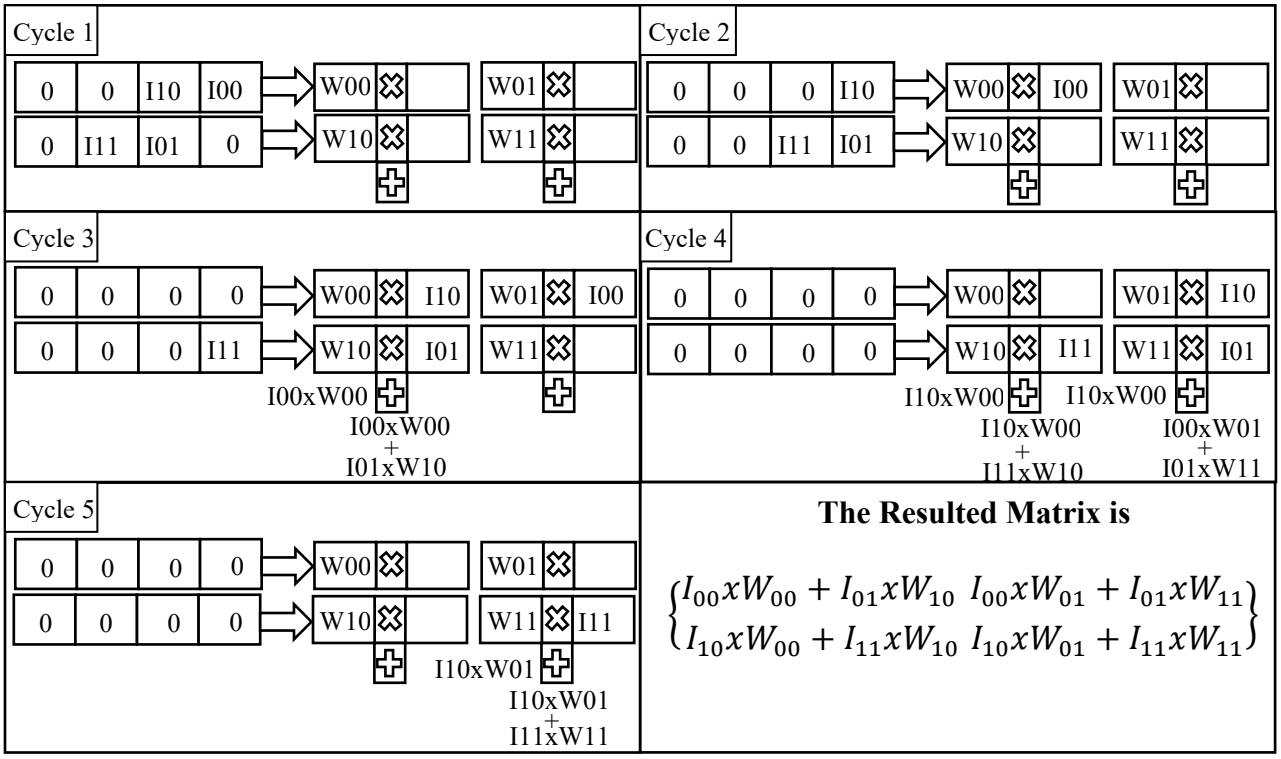


Figure 2-14 Matrix Multiplication using Weight stationary dataflow

2.2.1.2.1.3 OUTPUT STATIONARY

The last type is stationing the partial products in order to reuse them when accumulating. As usual the flow is quite similar to the ones explained above, which is shown in

Figure 2-15. It is observed that, the number of ALUs elements in this type is less than the other, which may not always be the case, the example given here is dedicated to this 2x2 Matrix Multiplication.

2.2.1.3 REAL LIFE AI ACCELERATOR EXAMPLES

In this sub-chapter, some accelerators implemented by companies will be briefly showed, in order to magnify intensively the power needed to process a neural network. However, not all of them using the same dataflow, or even strategy, each of them having the pros and cons, but what is common in all of them, they were all built on the premises to be able to process neural networks.

2.2.1.3.1 TPU

TPU(Tensor Processing Unit) is designed by Google, it is an ASIC NNP (Neural Network Processor), as many ASIC designs, they have less control units, which means in this case, it is only accelerating the matrix multiplication, while instructions fetching and scheduling will be left for the CPU for instance. It was firstly introduced in 2016 as TPUv1, but it has not gain as much popularity as versions v2 and v3. The most important part of the TPU is the matrix multiply unit, it is based on the weight stationary which was previously explained. In Figure 2-16-a), the block diagram of the TPU is shown, to show how components are interconnected.

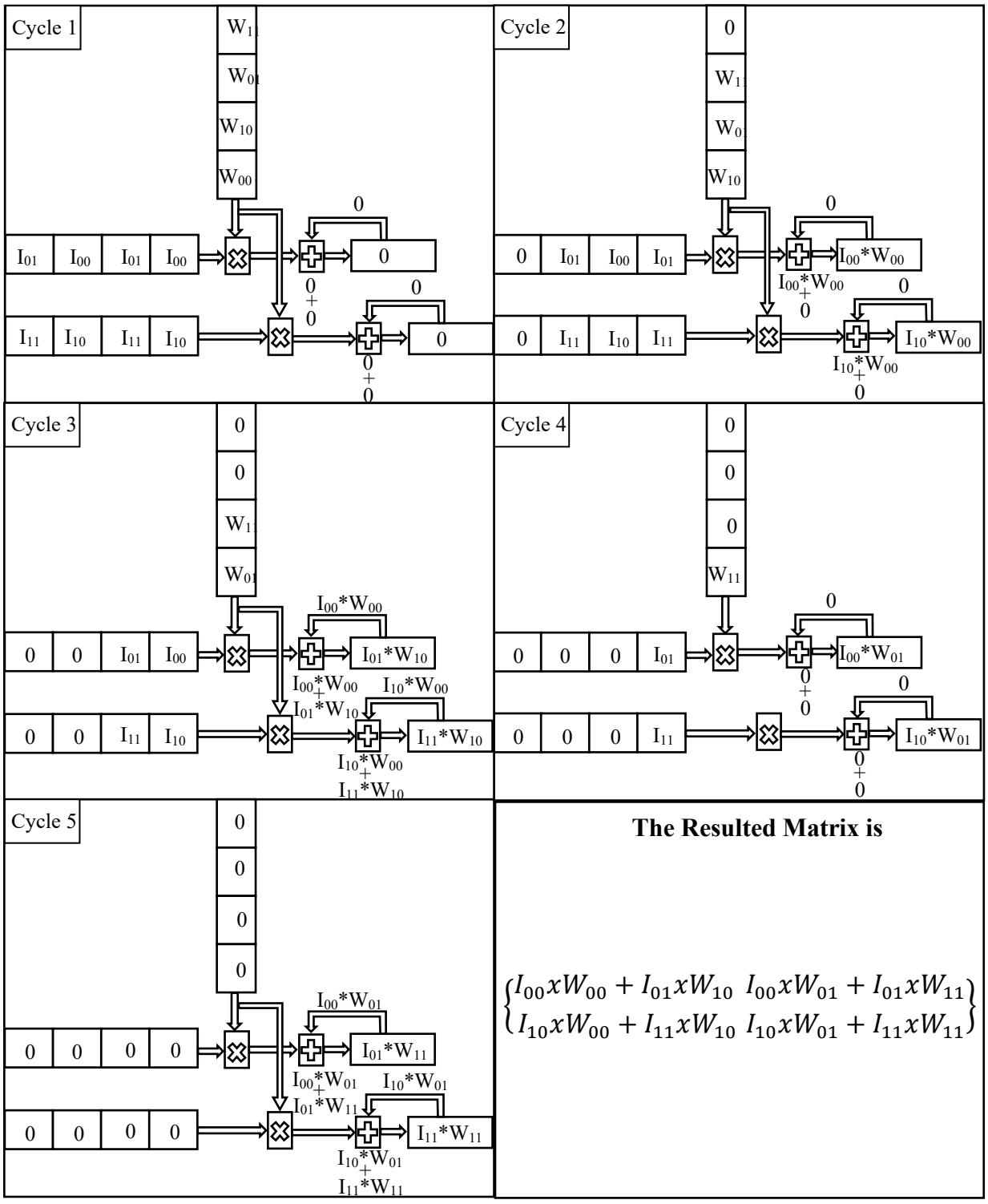
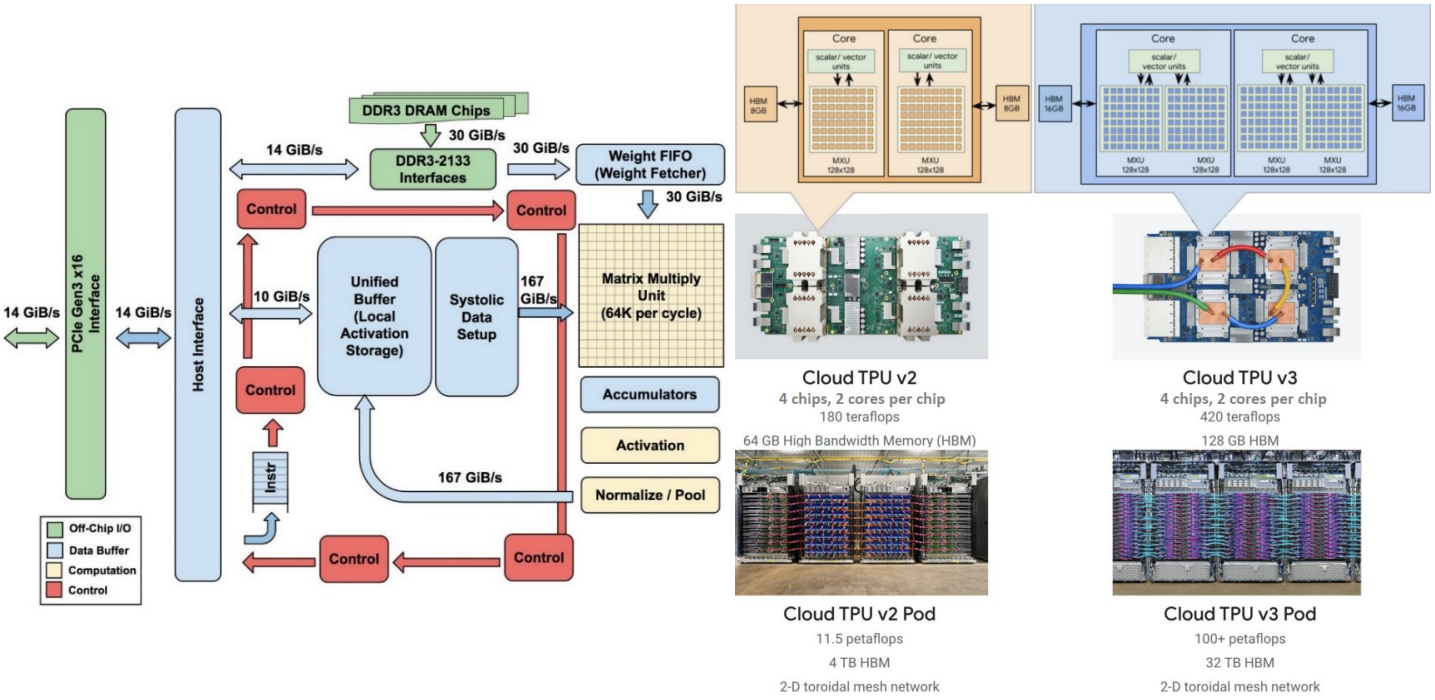


Figure 2-15 Matrix Multiplication using Output stationary dataflow



a) TPU block diagram, [27] **b) TPU versions cores differences, [28]**

Figure 2-16 TPU Architecture

2.2.1.3.2 EYERISS

Eyeriss is designed by MIT researchers and published in 2016. It is implemented based on TSMC 65nm LP technology. Like the TPU, it is using the strategy of reusing data. But unlike the TPU, it has control signals to choose which PE will be involved and has a very limited global buffer, in addition to it is not only using Weight stationary like TPU, it also has a taxonomy of collecting back the partial product sum and reuse it, this is called Row stationary, in Figure 2-17, the dataflow of Row Stationary is shown. However, the architecture of the Eyeriss is shown in Figure 2-18.

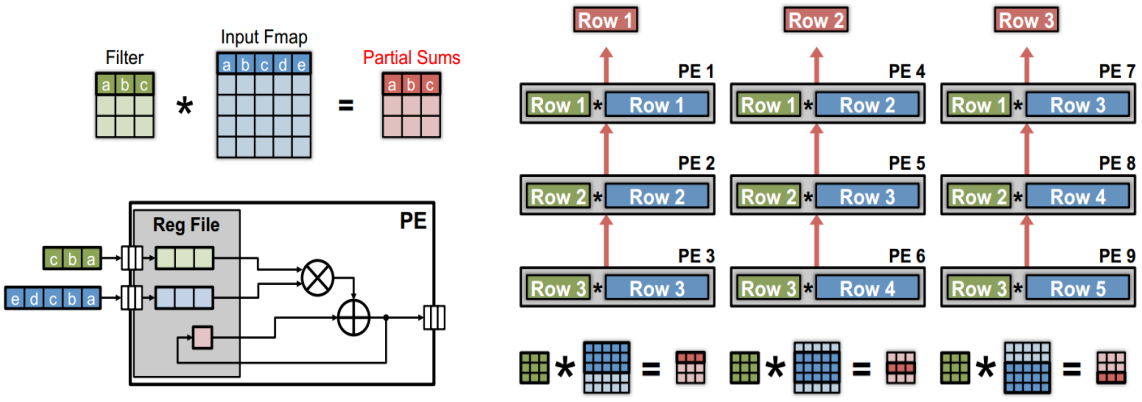


Figure 2-17 Eyeriss dataflow used, [24]

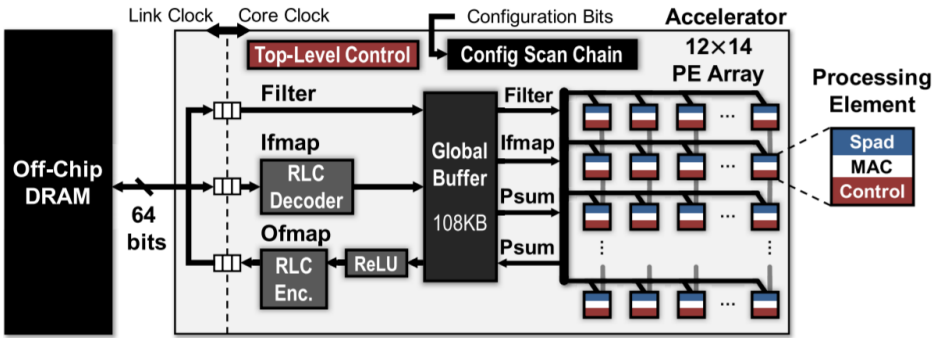


Figure 2-18 Eyeriss Architecture, [29]

2.2.1.3.3 NVIDIA GA100

Nvidia GA100 is designed by Nvidia and released in May 2020. It is implemented based on TSMC 7 nm. It is totally different architecture compared to the two showed above, in fact it is a very powerful GPU. But in addition to its different architecture, a Tensor core is embedded inside it, making it compatible to process very complex information, from video processing to low data processing. GA100 has 8 GPC (Graphic Processing Clusters) and in each one 16 SMs (Streaming Multiprocessors) inside it. In addition to that, it has 6 HBM2 (High Bandwidth Memory second generation) and for each one, there are two memory controllers, in Figure 2-19 this hierarchy is shown.

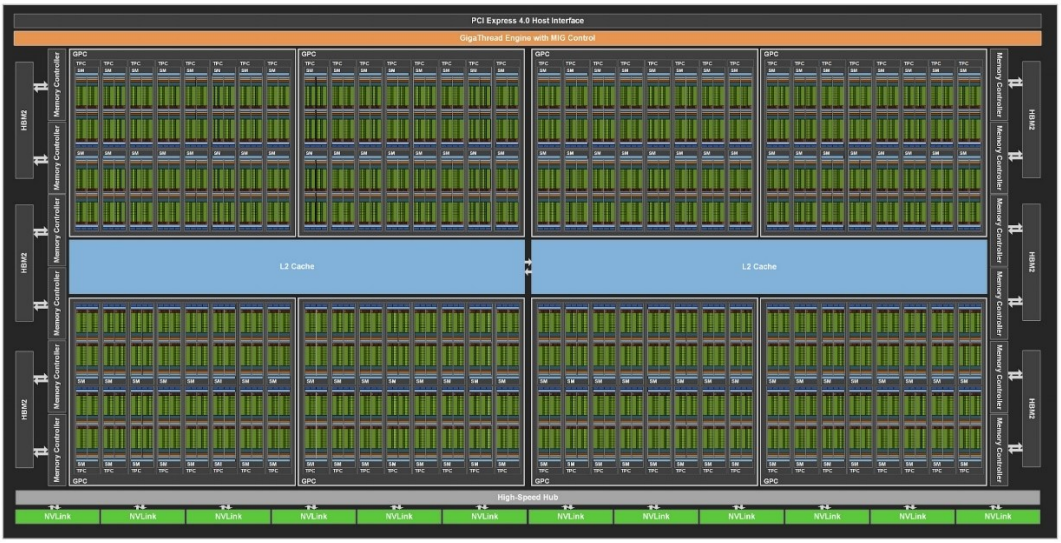


Figure 2-19 Nvidia GA100 Top-level Architecture, [30]

Each SM has: 4 processing blocks and 1 warp scheduler per processing block, 32 FP64 (64bits Floating points) CUDA (Compute Unified Device Architecture) Cores, 64 FP32 (32bits Floating points) CUDA Cores, 64 INT32 (32bits Integer) CUDA Cores, 192 KB of combined shared memory and L1 data cache and 4 tensor cores. The tensor core is not very different than the one implemented in Eyeriss which is shown in Figure 2-20. Another feature presented here is a new Sparse tensor core instruction that basically skips the weights with zero values, then it will be multiplied with the activation, as shown in Figure 2-21.

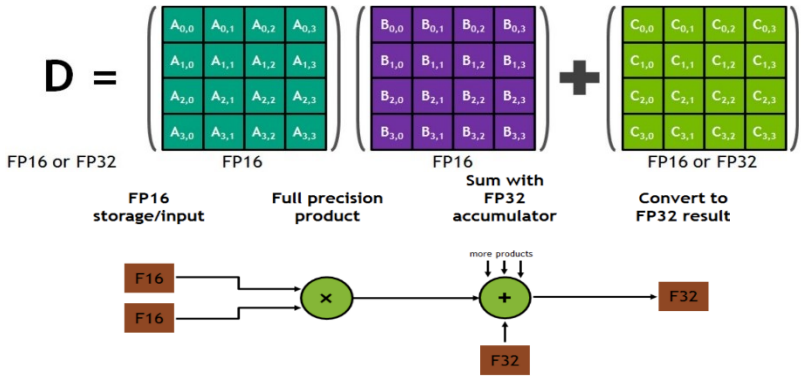


Figure 2-20 Nvidia GA100 Tensor core, [31]

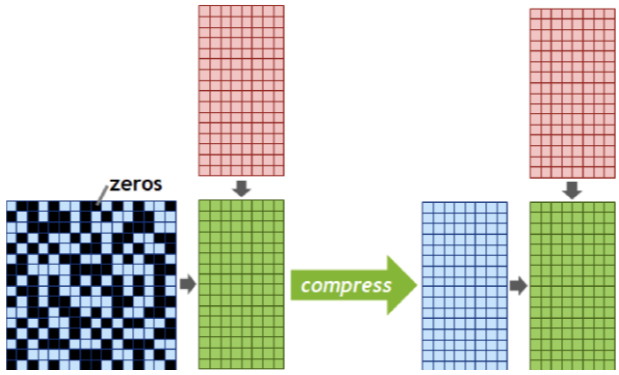


Figure 2-21 Nvidia GA100 Matrix Sparsity, [30]

2.2.1.3.4 INTEL STRATIX 10 FPGA

Intel Stratix 10 FPGA is designed by Intel in 2020, it is implemented based on intel 14nm tri-gate (FinFET) technology. As many companies face problems due to inflexibility of most of the processing units, they may tend to go with solutions like FPGA based designs, for the high configurability, even though, it is not very desirable for software Engineers. For instance, Microsoft Project Brainwave, [31], uses this FPGA to accelerate their DNN (Deep Neural Network). As one of the main drawbacks of FPGA the high-power consumption, which there is inadequate control over it, a specialized computational units called AI Tensor blocks are embedded inside, which is shown in Figure 2-22.

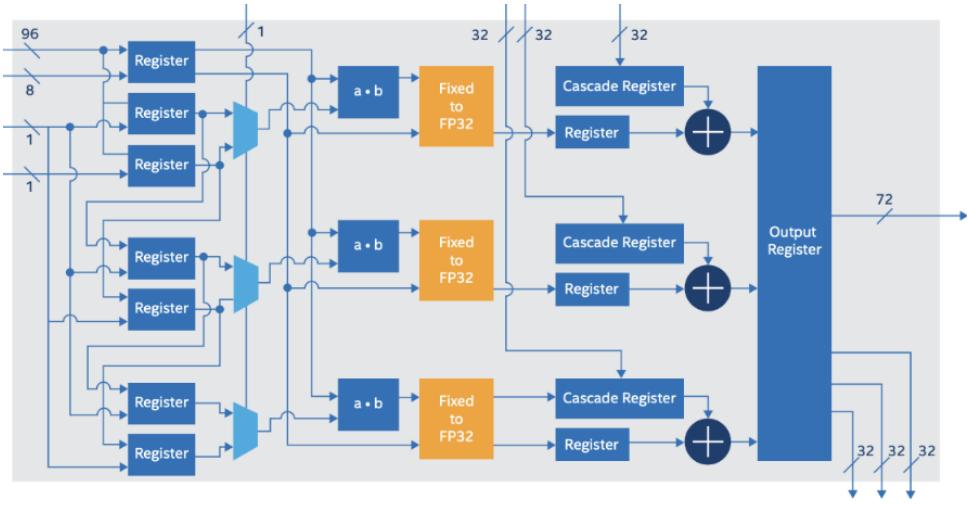


Figure 2-22 Intel Stratix 10 AI Tensor blocks, [32]

Since FPGA is programmed using HDL which are then compiled into bitstreams, these bitstreams not very compatible with the usage of DNNs. Therefore, in these bitstreams, DNNs are mapped into a pipeline in order to be supported to be processed. In order to understand this mapping, see Figure 2-23.

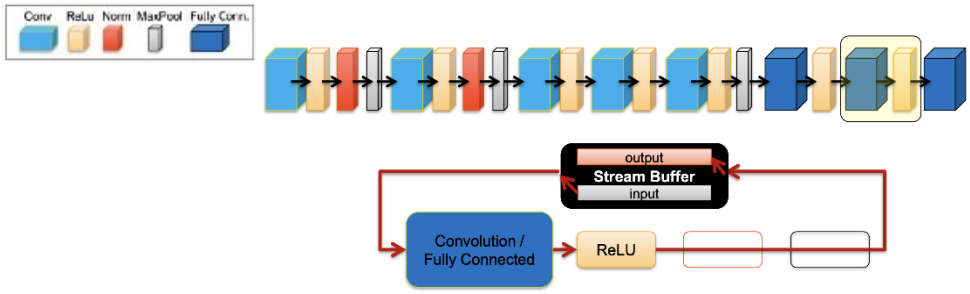


Figure 2-23 Neural network pipelining, [33]

Chapter three

3 NEURAL NETWORK ACCELERATOR DESIGN

Nowadays, the need of using machine learning is hugely increasing. Since technology is getting more complex day by day, it is necessary to adapt the neural networks used by machine learning algorithms nowadays. In order to implement those, especially in low scale applications, it is necessary to find another solution rather than *GPU*s (Graphical Processing Unit) which consumes much more energy and relatively consumes bigger area, than the design that is proposed in this work. Therefore, this work focuses on using weight stationary on a parametric and generic design of an accelerator using a new high-level HDL, with the ability of simulating, both the hardware designed when implementing a CNN on it, and verifying all the results after converting the CNN into a matrix multiplication.

As the most important issue in implementing neural networks on *GPUs* which is used by many scientists nowadays is its dataflow. The way *GPUs* process this network is not the best, in a sense that, most of them have a single very fast memory, which is in many cases built based on *DDR5* (Double Data Rate fifth generation) *DRAM* (Dynamic Random-Access Memory) technology. This memory is the only thing that the Execution unit, which is the *ALU* (Arithmetic Logic Unit) in this case can communicate with. In other means, when fetching out data, it has to come only from the *DRAM*, and if there are multiple data that need to be fetched, which is always the case, these data cannot bypass to the other *ALU*. This has a severe impact when it comes to the possibility of continuing using these data, instead of fetching them out each time from the *DRAM* it is possible to bypass them from the *ALUs* around it, but in this case, it is not only an *ALU*, it is a unit called *Processing Element*, that has an *ALU* integrated with a register file.

3.1 CHISEL

In this sub-chapter, the tools that were used to create this framework will be explained.

Since, the objective of this work, to design a parametric neural network accelerator. It was necessary to choose a hardware description language, that is capable to stand the complexity of this design. Also, easier to be used. High-level *Hardware description language* (*HDL*) allows the

usage of components like registers and logic shifters and many more, by just calling them, as a function, without the need of designing such these components. In fact, it is so close to *C* language. Allowing designers to write an abstract about the design in an algorithmic behavior, having the same characteristics of the classic *hardware description language* (*HDL*), then translates it to hardware digital design.

Generally speaking, high-level HDL is easier to use than classic *HDL*s. In a sense that, classic *HDL*s require designing all components that is needed for the top-level design. This could waste a lot of time when designing complex designs, such as this one. Besides, the poor capability of describing almost each single bit in the design. Classic *HDL*s can be better when designing architectures with very low scale, which has very small number of unknown components to *Chisel* library. In addition to the possibility to optimize the architecture to the maximum possible optimization, with very high controllability comparing to high-level *HDL* which usually call out components from the declared libraries it has.

Chisel is an example of high-level *HDL*, that was developed by University of *Berkeley*. The version used in this work is *Chisel3*. It is built over *Scala* language, that adds hardware capabilities. It is also giving the possibility of verifying the *HDL* by simulating it on the *Scala* code, by poking certain values into the hardware design, and check if the result is correct as expected or not. This gives the power for hardware designers to code, simulate and verify in a single environment.

3.1.1 SCALA BUILD TOOL

Scala build tool (*SBT*) is an open source tool, that is generally made for compiling *JAVA* and *Scala* programming language. Since *Chisel* is embedded in *Scala*, it used in this work to simulate the accelerator by creating a test file, that is written just as a programming language, that does not require any knowledge of hardware designing. This test file is used to create a virtual memory holding Pseudo random numbers with the neural network convolution layer parameters. In other words, it is just like simulating a real 2-dimensional convolution layer and test it on the *Chisel* code version with creating a virtual clock as well, and poking the input ports with the convolution layer data, and test the data coming out from the output ports and compare them to the code test

version. Above that, after verifying the results, it can create a *Value change dump* (*VCD*) that will be used in order to test the switching activating over the gates after synthesizing the design.

3.1.2 FLEXIBLE INTERNAL REPRESENTATION (FIR) FOR (RTL)

During compilation, the design passes by another compiler framework called *Flexible Internal Representation*, that optimizes this design to the same form of using classic *HDL*s. In this case, *Chisel* language, gives all the advantages that any hardware designer seeks, with the help of *FIR* too, it exports powerful performance with easy coding.

This was the reason behind, the selection of *Chisel* and all its platforms and tools, above all the other languages. When designing complex designs, generally designers care about:

1. Simplicity when writing code, to not waste time, about little small details, that is considered to be waste of time when designing complex designs.
2. Achieving very high performance, when transforming the behavioral level to circuit-level.
3. Ability of easily verify the hardware design, in an integrated form.

Clearly, all these requirements are fulfilled when using *Chisel* and all its supported tools

3.2 VERILATOR TOOLCHAIN

Finally, after verification and optimizing the design, it is time to synthesize it, in order to transform it into circuit-level. At this time, *Chisel* is not supported by industrial *EDA* tools. Fortunately, *Verilator* tool is supported by *Chisel*, which has the power to convert the design after being optimized into a netlist written in *Verilog* language, which is supported by almost all hardware design compilers.

3.3 DESIGN ANATOMY

After understanding all aspects of the convolution layer and the process taken for designing the accelerator as a functional level, until transforming it to circuit-level. It is possible now to create an architecture suitable for all aspects explained previously.

In this work, a spatial architecture based on the weight-stationary systolic paradigm has been implemented. The reason behind using this kind of dataflow is, the ability of saving power, by keeping Weights data static and reuse them, up to the maximum possible number of uses. Also, the ability of shifting inputs all along the array, without the need to access the memory each time an input data is needed

The following sections will explain the design from top down analogy.

3.4 CONVOLUTION LAYER ANATOMY

Designing an accelerator for neural networks requires special dataflow, since neural networks have quite big set of data frames, that must be computed in a synchronized form. However, the focus here is on convolutional layers because they represent the most computationally intensive part of the model execution. Moreover, Convolutions layers in a neural network, for instance, do not always have same set of hyperparameters. Therefore, designing an accelerator, should be able to sustain different dimensions. Nevertheless, the layer is considered with certain parameters, which are:

1. Input shape (C_{in}, H_{in}, W_{in}): Input shape represents the image that need to be processed. Usually, this image is represented by number of rows, number of columns, which are the matrix of an image, which represents number of pixels of an image, and set of colors for this image, like RGB, which are number of channels. Each channel will be filtered by n number of filters, with a specific dimension. This was a dedicated example when processing an image as the input shape. However, each case has its own criteria, but in general it is composed of rows, columns, and channels.

2. Output shape (C_{out}, H_{out}, W_{out}): Output shape represents, product of image channels after being filtered. It is possible that, in some convolution layers, the number of output pixels and input pixels, do not match and This is because of the stride.

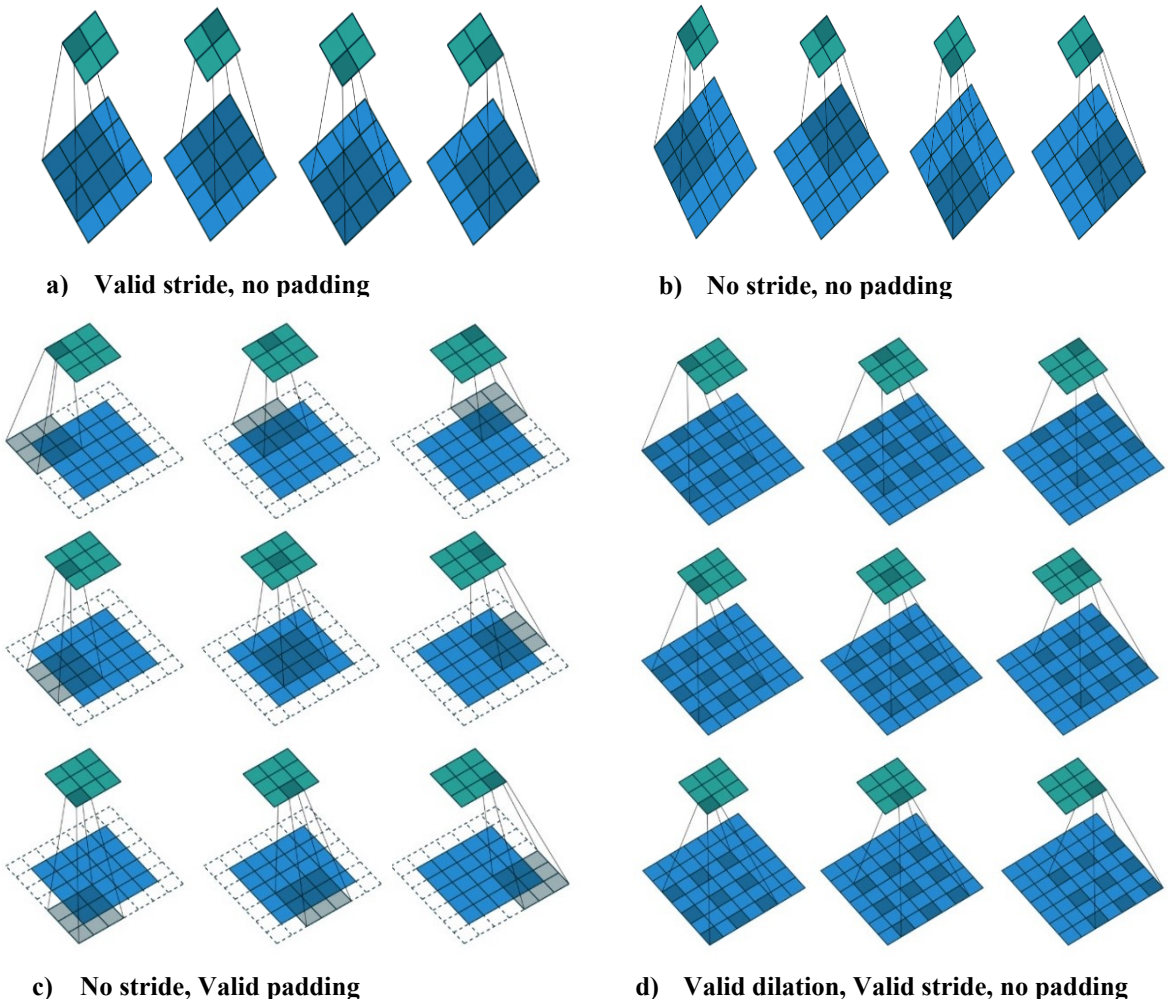
3. Padding: This is mainly used, to maintain the input dimensions, in output dimensions, by adding m rows to the input, one placed in the first row and one in the last row, also, adding column in the beginning and in the last, and place certain value in them.

4. Strides: This is used to shift the window of the kernel number of times, usually, 1 or 2 times, along the input shape.

5. Dilations: It is used to skip a pixel in the input, while filtering it, using the kernel

6. Kernel shape (C_{out}, C_{in}, K_x, K_y):

The below figures can help in understanding these parameters much more.



a) Valid stride, no padding

b) No stride, no padding

c) No stride, Valid padding

d) Valid dilation, Valid stride, no padding

Figure 3-1 Convolution layer parameters visualization, [34]

Since the proposed architecture, performs a matrix multiplication operation, it is essential to rewrite any convolution layer as a matrix multiplication. In order to do so, the image is converted to column using one possible way which is *im2col* strategy. Figure 3-2 is explaining this strategy.

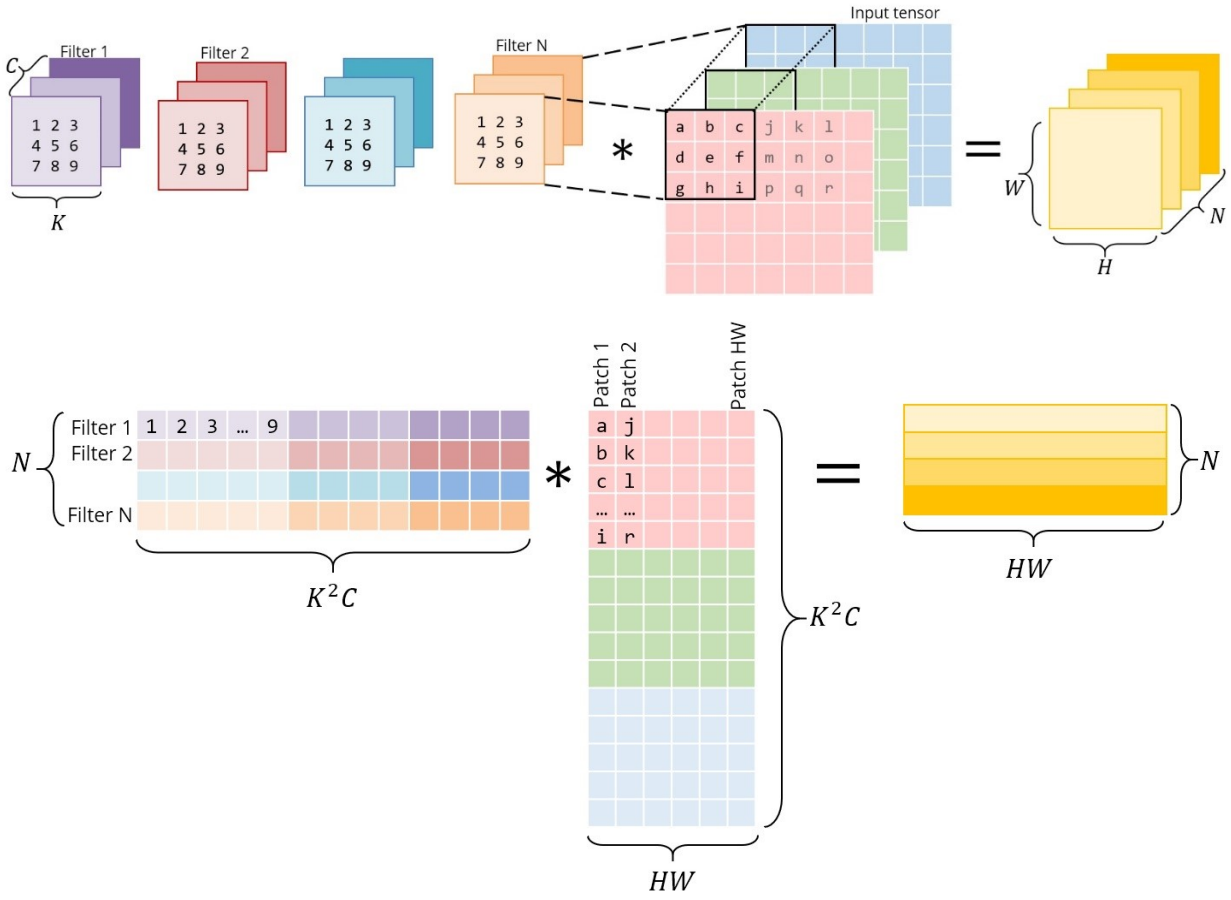


Figure 3-2 *GeMM im2col* strategy, [35]

Filters is representing the kernel shape, while, input tensor, represents the input shape, that was previously explained. Clearly, the values of padding, stride and dilation faded out with this strategy.

Last step is transposing these matrices. Since the strategy used is *Weight stationary*, it is important to have Matrix B is the weight and Matrix A is the input, exactly the inverse of Figure 3-2. Hence, Matrix A which refers to the extension of the Input tensor, will have a dimension of (input height*input width, Kernel height * Kernel width*Input channels) and Matrix B which refers to the filters that are applied on the input tensor, which is called Weights after being transformed, will have a dimension of (Kernel height * Kernel width*Input channels, Output channels). This shall result an output of dimension of (input height*input width, Output channels), but since there might be a stride, the output row can change.

3.4.1 PROCESSING UNIT DESIGN

Inside the *Processing Unit*, there are couple of components, that are all connected to an external memory. these components are:

1. *Processing Array* (*PA*): it is a term called, which defines a group of blocks called *Processing Element* (*PE*). In this work, these blocks will process the Weights and Inputs of the convolution layer. In most cases, this *PA* has a dimension which is way smaller than the convolution layer dimension. Here comes the ability of processing of huge dimensional data, with few resources.

2. Buffers: There are two group of buffers inside this *PU*, one group is for inputs data, and the other for partial products.

Figure 3-3 shows the top view of what does this *PU* looks like.

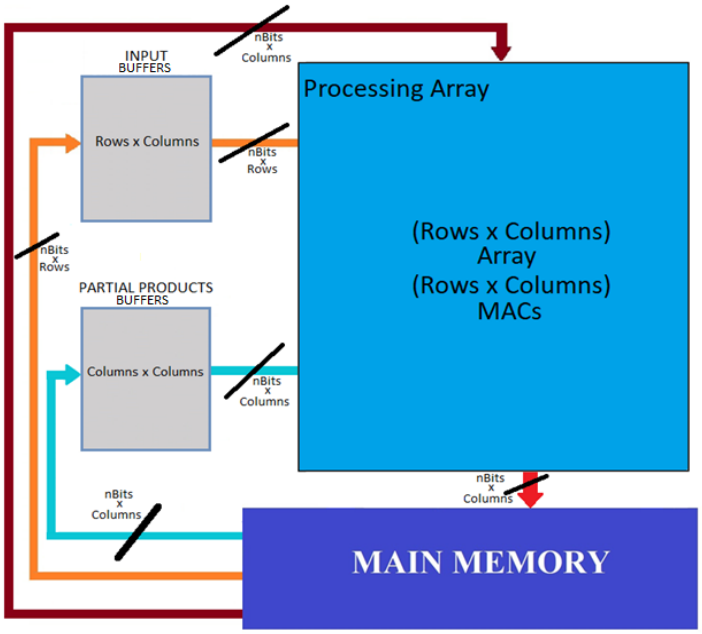


Figure 3-3 Top view of the *Processing Unit*

It is important to mention that, the main memory in Figure 3-3, is not inside the *PU*, but since it stores the data of the convolution layer, it was essential to show it up.

The design has 3 input ports and one output port, each port is made of n number of bits. One input port for Weights, which has a number of access ports equals to the number of columns of the *PA*, since Weights will shift right into the *PA* in all columns at the same cycle for number of cycles equal to the number of rows. This means that the last row will be filled at the end of data organizing. Unlike all the other input ports, Weights does not require to be stored outside the *PA*, since they will be used only once, and there is no need for shifting them after the organization cycle .The second input port is for data inputs, which will be stored inside the shift registers until the organization cycle ends. The last input port is made for partial products, which has the same behavior of the data input port.

All inputs and outputs ports are connected directly to the main memory. Which has the weights and inputs data of the convolution network.

The design acceleration can be classified into different types of cycles:

1. General cycle: This cycle will contain all the cycles mentioned below. The number of iterations in this cycle is equal to:

$$Ceil\left(\frac{Matrix_{B_{Row}}}{PA_{Row}}\right) * Ceil\left(\frac{Matrix_{B_{Column}}}{PA_{Column}}\right)$$

Equation 3-1 Number of iterations calculation in the general cycle

   In each cycle of that, a window of Weights will be taken. The dimension of the window is always equal to the *PA* dimension.

   a. Organization cycle: It is the first cycle which deals with taking a window of the weights, and place them inside the *PA*. Usually, a single window is taken after passing by all the cycles in the general cycle. The window starts from the top left of the Weights 2D array going to the last row, then shift right by the number of columns and start again from the first row. At the same time, the inputs shift inside the input buffers, for number of cycles equal to the *PA* number of columns. This means that the column of the input buffer in the very right, which is considered to

be the first column, will be filled then it will stop receiving input data from the memory during this cycle.

The number of iterations for this cycle can be calculated by using this equation:

$$Max(PA_{Row}, PA_{Column})$$

Equation 3-2 Number of iterations calculation in the organization cycle

Consider having a *PA* of dimension 3x2, weight of 3x2 dimension and input of 2x3, in order to understand this cycle.

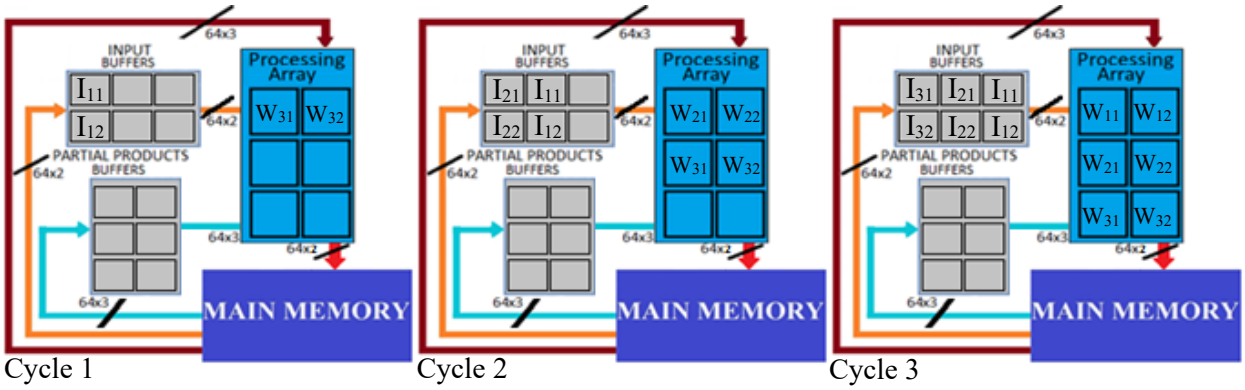


Figure 3-4 Organization cycle sample

During this cycle, no inputs will enter the *PA*, and here, this cycle ends with iteration equal to 3, which is equal to the number of columns.

b. Data processing cycle: inside this cycle, Weights will not change, but inputs data inside the will keep shifting inside the *PA*. In addition to that, there will be two inner cycles inside, which will be processed at the same cycle:

    i. Input shifting cycle: In order to use the Weights, the maximum number of usages, the buffer will receive all the inputs inside number of columns, which will be equal to the columns of the *PA*. Continue the example of Figure 3-4, but focusing only on the Input buffer part.

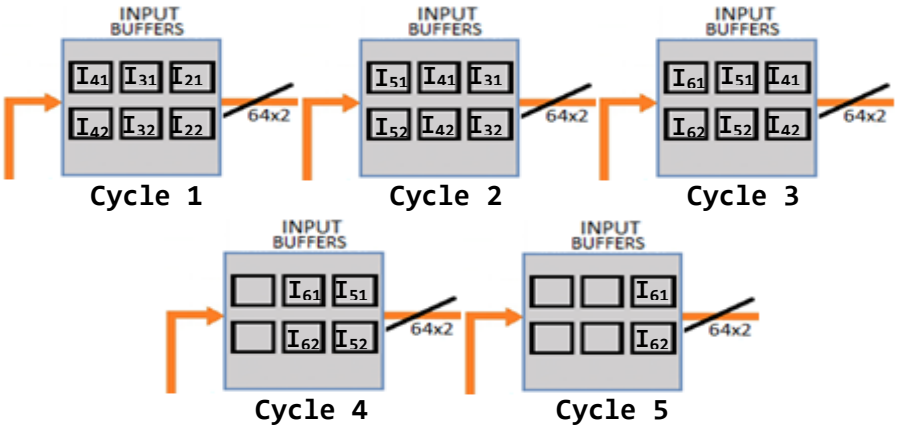


Figure 3-5 Input shifting cycle sample

After cycle 5, the *PA* will have all the inputs and weights to be processed. Right after cycle 5, the number of iterations left to finish data processing cycle is equal to:

$$(PA_{Row} - 1) + (PA_{column})$$

Equation 3-3 Number of iterations calculation in input shifting cycle

ii. Partial products shifting cycle: The purpose of this cycle, is accumulating the results of the previous general cycle with the current general cycle, since, as mentioned before, this accelerator is mainly used with convolution layers with big dimensions. This cycle process will be explained in further details in the Partial products accumulation section. However, this cycle does not really start in the beginning of the data processing cycle. In fact, it starts after the processing cycle equals to:

$$PA_{Row} - 1$$

Equation 3-4 Starting condition of Partial products shifting cycle

Nevertheless, this cycle is executed in parallel with the input shifting cycle.

The number of iterations for the data processing cycle can be calculated using this equation:

$$(PA_{Row} + PA_{Column}) + Matrix_{A_{Row}} - 1$$

Equation 3-5 Number of iterations calculation in Partial products shifting cycle

Now, after splitting down all the cycles, it is possible to calculate the total number of iterations by:

$$General\ cycle * (\text{Organization cycle} + Data\ processing\ cycle)$$

Equation 3-6 Total number of iterations calculation

After each general cycle, all the buffers and register files receive reset signal, so they no longer hold any data inside.

For better understanding, the waveform in Figure 3-5 can show how data is synchronized, knowing that, the example used in this figure, is the same one proposed in Figure 3-4.

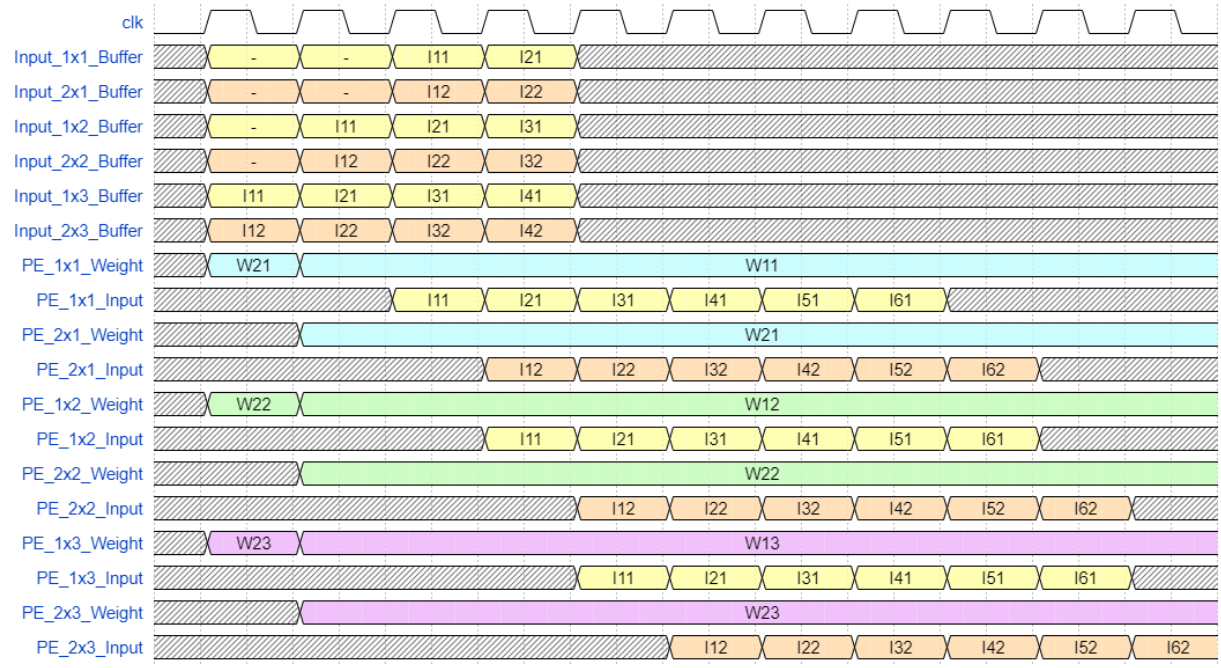


Figure 3-6 Processing Unit synchronization waveform

The reason why the input data in each row, are not showing up in the same cycle, is that there are internal registers inside the *Processing unit (PU)* that keep the data in the right synchronization. Further details about this part will be explained in Internal registers section.

3.4.1.1 PROCESSING ARRAY

This is the most important component in the *PU*, since all the computations occur in it. This block is made up of several essential blocks, called *Processing engine* and some registers. The importance of these registers is to delay some input and output data from another, so they shall be synchronized. Another importance is, to shift data inside the register files of the *PE* all along the other register files.

The processing array is divided into two groups:

1. Systolic array group: it deals the processing inputs and weights of the convolution layer, in a systolic movement. This group is connected to the input buffers and the main memory, for shifting the Weights inside the *PE*s.

2. Accumulator group: it accumulates all the partial products all together to give the correct output product, in the very end. This group is connected with the output of the first group, which is the systolic array group and the partial products buffer.

A general view for this division is in Figure 3-7.

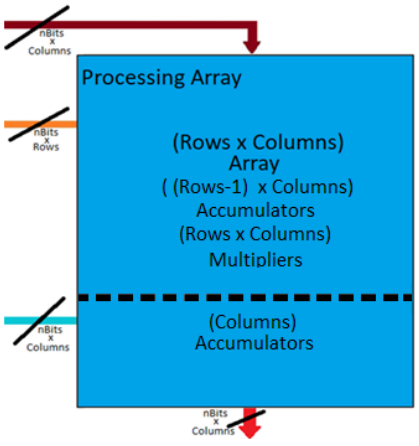


Figure 3-7 Processing Array top view

Consider a simple matrix multiplication of A (4, 4) and B (4, 4), and model it on the proposed *PA* design, with dimension (2, 2).

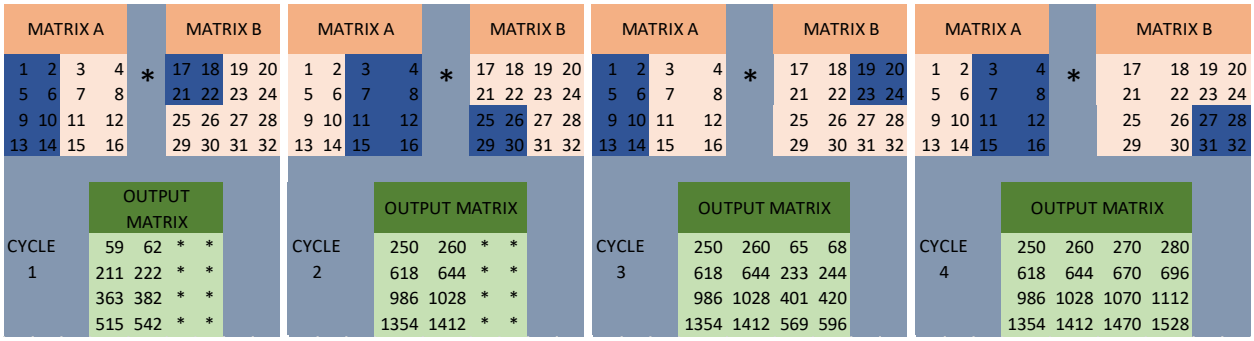


Figure 3-8 Matrix multiplication sample

The cycle in Figure 3-8 is referred to the general cycle, which is explained in Processing unit design. However, as shown in the figure above, Matrix B was windowed 4 times, upon those windows, the values were used once. Unlike Matrix A, which the values of it, were used more than once, like in cycles 1 and 3. However, the above figure is representing the dataflow strategy inside the *PA*. In this particular example, Matrix A is representing Input data and Matrix B is representing the Weights. Also, the accumulation strategy was represented in cycles 2 and 4, in which the values of cycle 1 was added to the values that were calculated in cycle 2. And the same implies between cycle 3 and 4.

This is exactly the kind of behavior aimed to be implemented in this *PA*.

3.4.1.1.1 PROCESSING ELEMENT

PE is generally made of Register file, that is divided into two parts and Multiply and Accumulator (*MAC*). The first part of the Register file is assigned for the Input data, and the second part for the Weights. Usually the number of *PEs* inside the *PA* is equal to the assigned values of Rows and Columns of the *PA*. Since this is a matrix multiplication at the moment, the only needed operands are '+' and '*'. In Figure 3-9, the block of *PE* is shown.

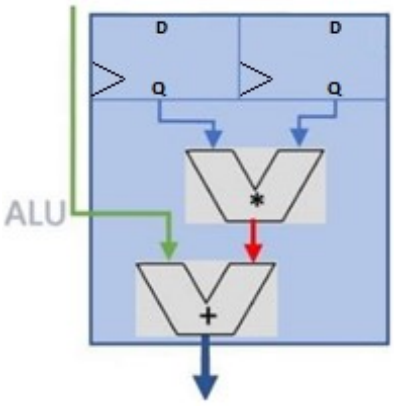


Figure 3-9 Processing Element Top view

Usually, only the first row of the *PEs* looks like Figure 3-10. It does not have adders, since the first row will have any values to accumulate.

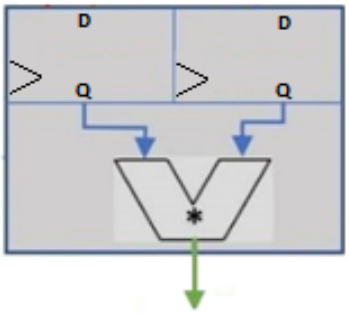
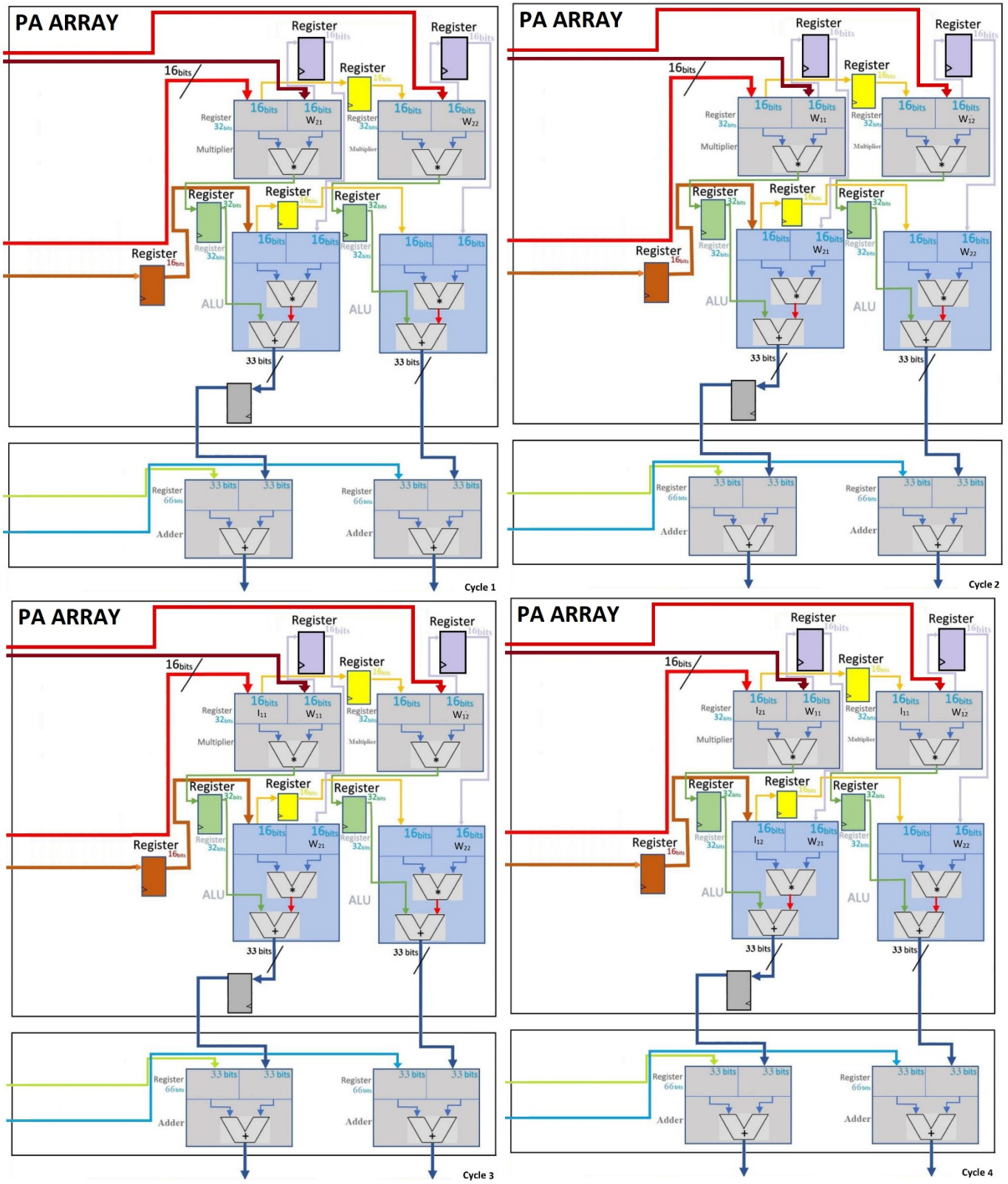


Figure 3-10 Processing Element top view of the first row of the Processing Array

Let us consider cycle 1 in Figure 3-8, with using the proposed hardware components of 16 bits.

In Figure 3-11, cycles 1 and 2, are considered to be Organization cycle, while the rest, are considered to be Data processing cycle. Cycles 3 to 6 are Inputs shifting cycles are activated, and after that it is disactivated.

The accumulation effect did not appear in this example, since only the first cycle of the general cycle was implemented, and there were no partial products.


PA ARRAY
Register
16bits
Register
16bits
Register
16bits
16bits
16bits
W21
Register
32bits
Multiplier
16bits
16bits
W22
Register
32bits
Multiplier
Register
32bits
Register
16bits
Register
32bits
Register
16bits
16bits
16bits
Register
32bits
Register
32bits
ALU
16bits
16bits
ALU
16bits
16bits
33 bits
33 bits
Register
66bits
33 bits
33 bits
Adder
Register
66bits
33 bits
33 bits
Adder
Cycle 1
PA ARRAY
Register
16bits
Register
16bits
Register
16bits
16bits
16bits
W11
Register
32bits
Multiplier
16bits
16bits
W12
Register
32bits
Multiplier
Register
32bits
Register
16bits
Register
32bits
Register
16bits
16bits
16bits
Register
32bits
Register
32bits
ALU
16bits
16bits
W21
ALU
16bits
16bits
W22
33 bits
33 bits
Register
66bits
33 bits
33 bits
Adder
Register
66bits
33 bits
33 bits
Adder
Cycle 2
PA ARRAY
Register
16bits
Register
16bits
Register
16bits
16bits
16bits
I11
W11
Register
32bits
Multiplier
16bits
16bits
W12
Register
32bits
Multiplier
Register
32bits
Register
16bits
Register
32bits
Register
16bits
16bits
16bits
Register
32bits
Register
32bits
ALU
16bits
16bits
W21
ALU
16bits
16bits
W22
33 bits
33 bits
Register
66bits
33 bits
33 bits
Adder
Register
66bits
33 bits
33 bits
Adder
Cycle 3
PA ARRAY
Register
16bits
Register
16bits
Register
16bits
16bits
16bits
I21
W11
Register
32bits
Multiplier
16bits
16bits
I11
W12
Register
32bits
Multiplier
Register
32bits
Register
16bits
Register
32bits
Register
16bits
16bits
16bits
Register
32bits
Register
32bits
ALU
16bits
16bits
I12
W21
ALU
16bits
16bits
W22
33 bits
33 bits
Register
66bits
33 bits
33 bits
Adder
Register
66bits
33 bits
33 bits
Adder
Cycle 4

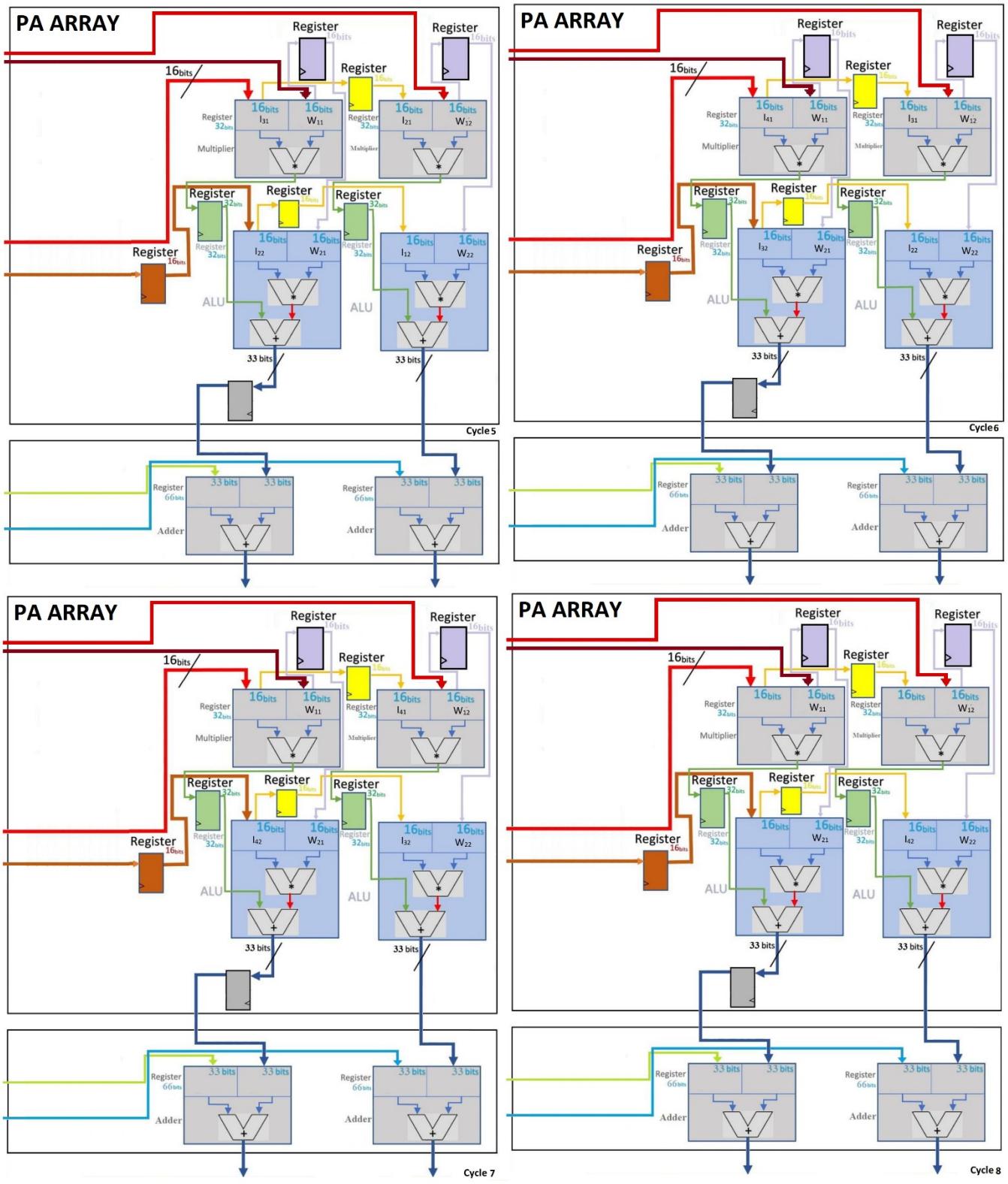


Figure 3-11 Sample of processing a Weight (4x4) with Input tensor (4x2) in Processing Array with hardware components

3.4.1.1.2 INTERNAL REGISTERS

From Figure 3-11, it is clearer now, the importance of the internal registers. But let us divide those registers into five categories, according to their functions, referring the colors of the registers to Figure 3-11.

1. Orange registers: Delay the inputs from entering the *PE* register-file to be correctly accumulated with the values of their previous rows. In order to correctly synchronize the accumulation, the number of registers will increase by 1 register in each row, starting with no registers in the first row. The below equation can be used in order to calculate the total number of this register.

$$\frac{(PA_{Row} - 1) * PA_{Row}}{2}$$

Equation 3-7 Number of Input buffer calculation

2. Green registers: It is used to shift the values coming out of the *MAC*/*ALU* to the upcoming *PE MAC*/*ALU*. The below equation can be used in order to calculate the total number of this register.

$$(PA_{Row} - 1) * PA_{Column}$$

Equation 3-8 Number of Processing element buffer

3. Yellow registers: It is used to shift the input data inside the register-file to the register-file of the *PE* next to it. The below equation can be used in order to calculate the total number of this register.

$$(PA_{Column} - 1) * PA_{Row}$$

Equation 3-9 Number of Input shifters buffer

4. Violet registers: It is used to shift the Weight data inside the register-file to the register-file of the *PE* next to it. The below equation can be used in order to calculate the total number of this register.

$$(PA_{Row} - 1) * PA_{Column}$$

Equation 3-10 Number of Weight shifters buffer

5. Grey registers: Delay the outputs from entering the Accumulator part to be correctly accumulated with the values of the partial products. In order to correctly synchronize the accumulation, the number of registers will increase by 1 register in each column, starting with no registers from the last column. The below equation can be used in order to calculate the total number of this register.

$$\frac{(PA_{Column} - 1) * PA_{Column}}{2}$$

Equation 3-11 Number of Output shifters buffer

3.4.2 PARTIAL PRODUCTS ACCUMULATION

In each general cycle, there are partial products with dimension of $(Matrix_{A_{Row}}, PA_{Column})$. These products must be accumulated with other partial products, until the *PA* window reaches the last row of Matrix B $_{\text{Row}}$. While the actual dimensions of the partial products buffers will always be the inverse of the upper part of the processing array.

By looking at Figure 3-12, the blue blocks in Matrix A will be multiplied by the blue blocks of Matrix B, in each particular cycle. Thereby, the cycles in each group will be accumulated all together to give the final correct result. Since in this particular example, the window of the *PA* has a dimension of (2, 2), and number of groups are 4, so, the final output will give a dimension of (4, 2, 2). By extending this three-dimensional matrix, the result will be (8, 8), which is the correct dimension of the output.

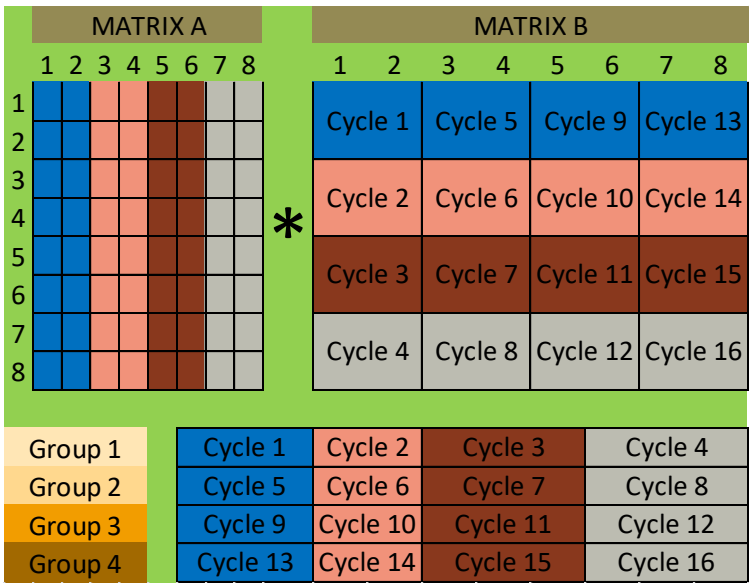
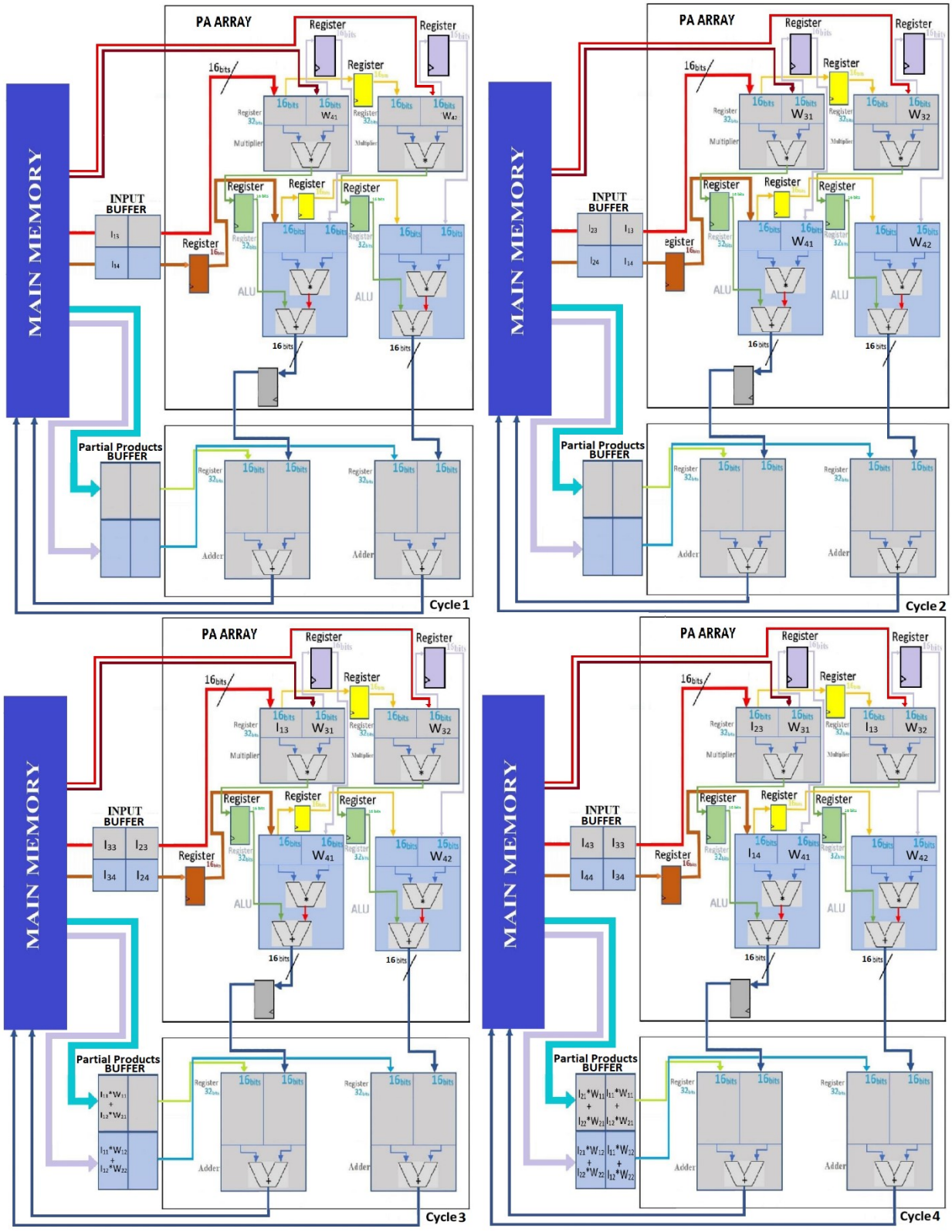


Figure 3-12 Sample of accumulating Weight (8x8) with Input tensor (8x8)

Let's complete the example of Figure 3-11, in Figure 3-13 using hardware components, by stepping to the general cycle after it, which is general cycle 2.


MAIN MEMORY
PA ARRAY
Register
16bits
Multiplier
INPUT BUFFER
ALU
Partial Products BUFFER
Adder
Cycle 1
Cycle 2
Cycle 3
Cycle 4

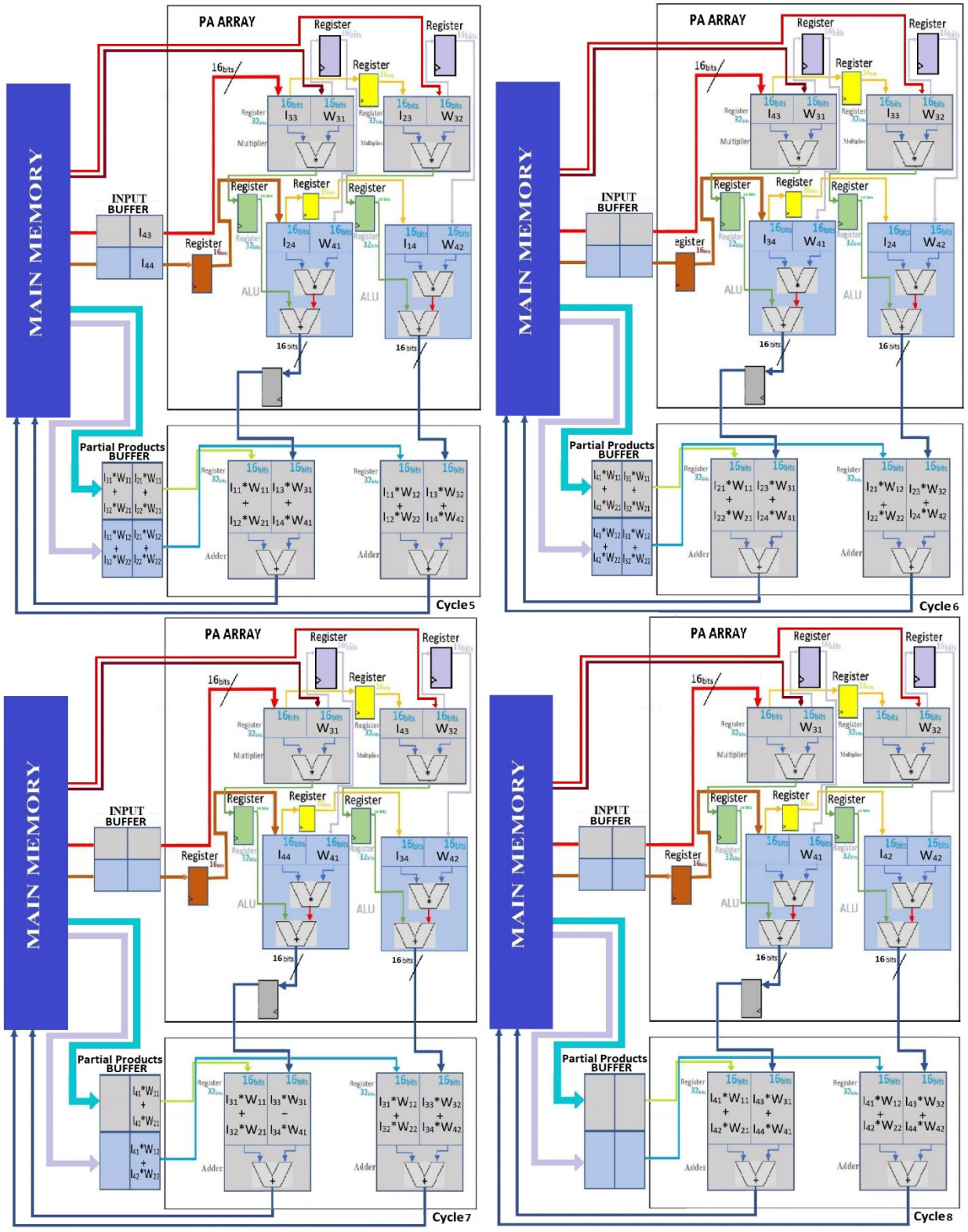


Figure 3-13 Completion of processing and accumulation a Weight (4x4) with Input tensor (4x4) in Processing Array with hardware components

To have the full output, these iterations will be repeated for four cycles, which are referred to the general cycles.

3.5 SYSTEM DESIGN

After understanding the design anatomy, and how to implement it, it is possible to start writing the code, both, the Hardware description language that is written in *Chisel* in this work, and its simulator, which is written in *Scala* in this work.

3.5.1 SCALA CODE

In order to have an organized environment and to make sure that the hardware accelerator will perfectly work and give correct result, a simulator was established first, with exactly the same analogy of the design anatomy.

Firstly, the neural network environment is defined, then, two virtual memories bank were created that holds both input tensors and kernels data as shown in the below flowchart.

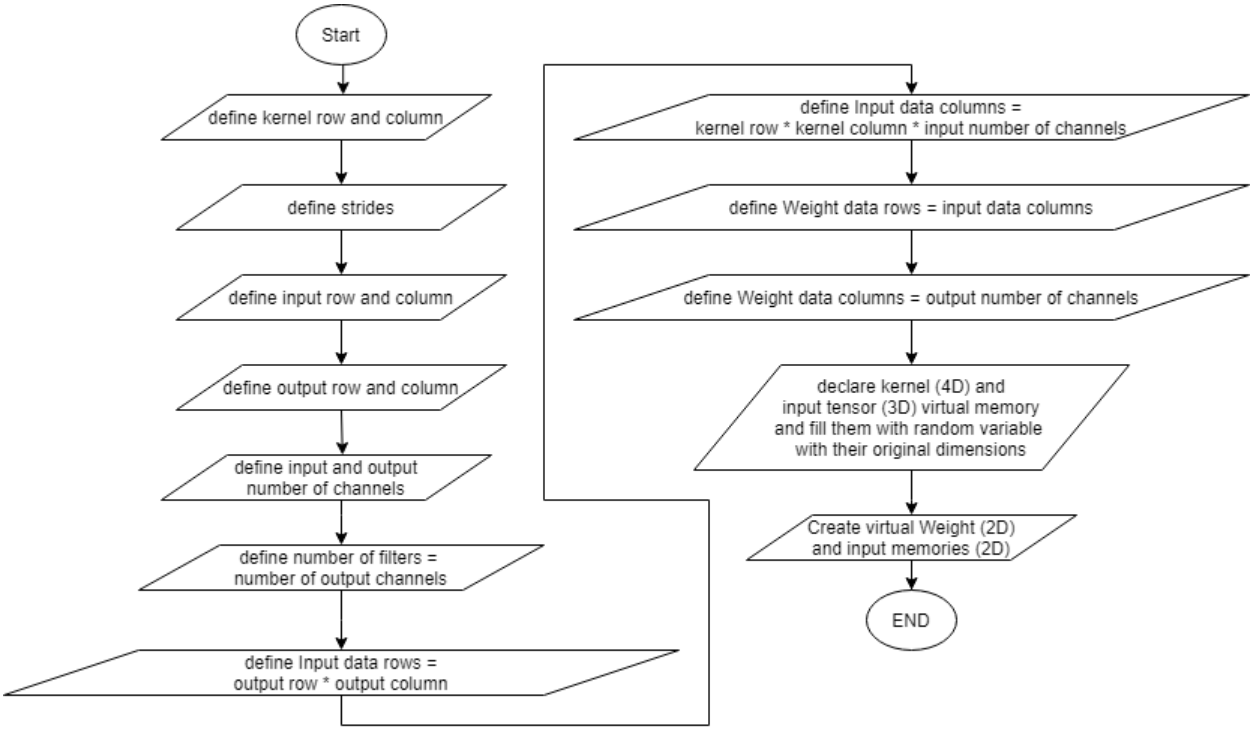


Figure 3-14 Neural network parameters definitions flowchart

These data are then distributed among the virtual memories, the kernels are inside the virtual weight memory, while the input tensor data are inside the virtual input memory. As illustrated in Design Anatomy chapter after being organized as *General matrix to multiplication* as *im2col* strategy was explained.

The distribution of the kernels is shown in the below flowchart.

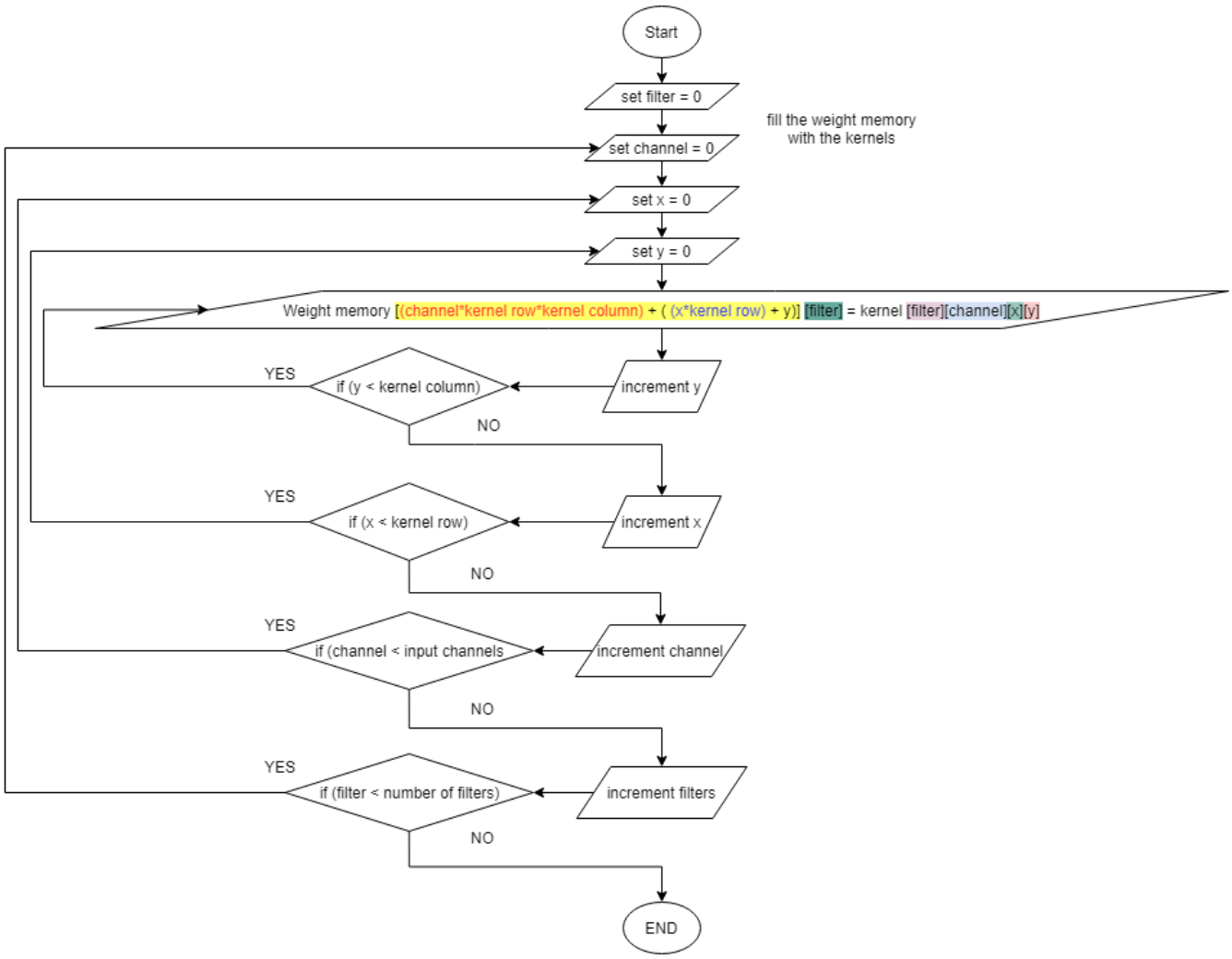


Figure 3-15 Kernel distribution flowchart

Now, the last memory distribution left is the input tensor, which is shown in the below flowchart.

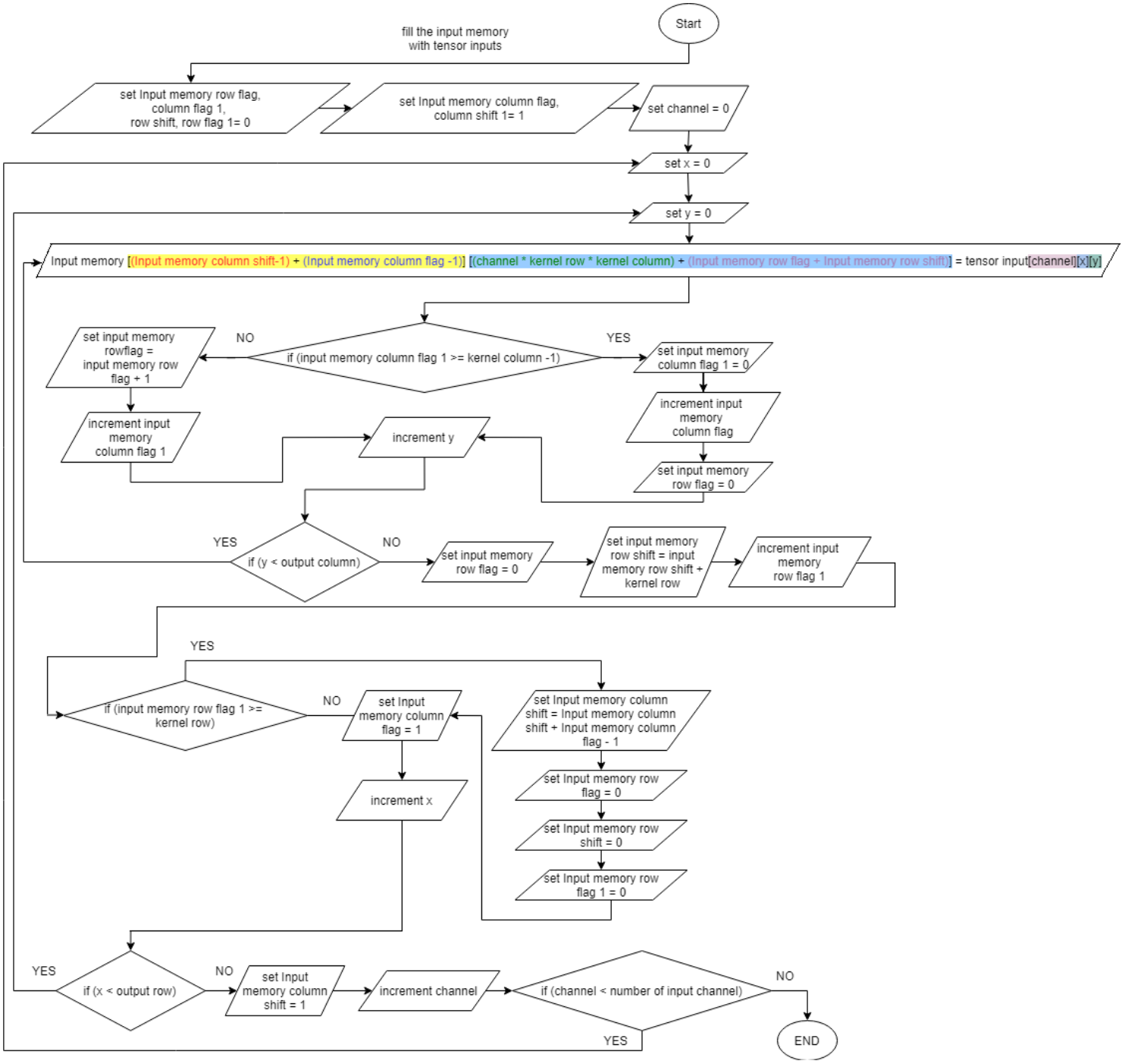


Figure 3-16 Input tensor distribution tensor flowchart

After having all data and memory correctly established, it is possible to establish the dataflow of the processing array.

First thing done, was calculating the aspects of the processing, which are the number of cycles required and number of iterations, as explained in Processing unit design sub-chapter. Right after, the internal parameters of the processing array are declared, which are the internal register and the multiply and accumulator elements. These declares are shown in the below flowchart.

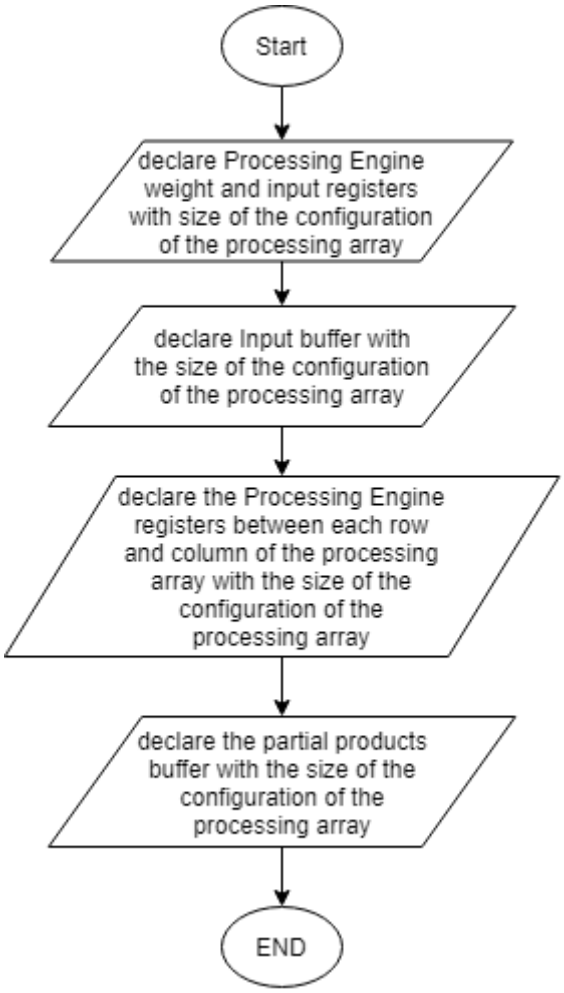


Figure 3-17 Processing aspects declaration flowchart

After having all physical components declared and data are organized in the virtual memories, it is possible to process them now.

For simplicity, the code will break down into:

1. General cycle

a. Data placement
    i. By row
    ii. By Column
b. Data processing
    i. Processing elements data shifting
    ii. The upper processing array part
    iii. The accumulator part
c. Data Resetting

In the general cycle stage, as explained before, it is the cycle which will have the weights inside the Processing array stationary and not changeable, in order to maximize its usage. Inside this cycle, there are smaller cycles. The below flowchart represents this stage.

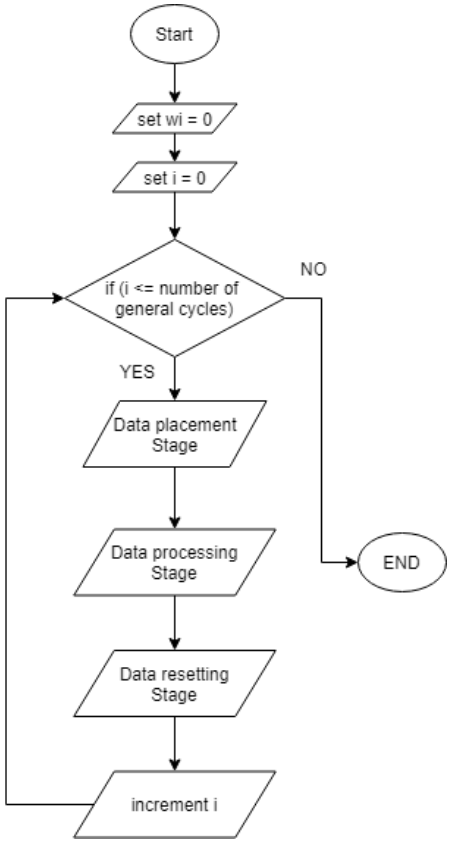


Figure 3-18 General cycle flowchart

Data placement stage will deal with taking data input from the virtual input data, and place some and maybe all of them inside the input buffer, it depends on the size of the processing array. Another thing occurs inside the data placement stage is, placing some of the weights and maybe all of them inside the processing elements weight register. This stage is divided into two stages, since this design is generic and parametric, it is possible that the number of columns of the processing array are bigger than the rows, so, in order to make sure that data are organized and filled correctly, it is checked by number of rows firstly, and then iterated to the number of columns, in case the columns are bigger than rows, otherwise, the columns stage is unnecessary. The below flowchart represents the Data placement stage.

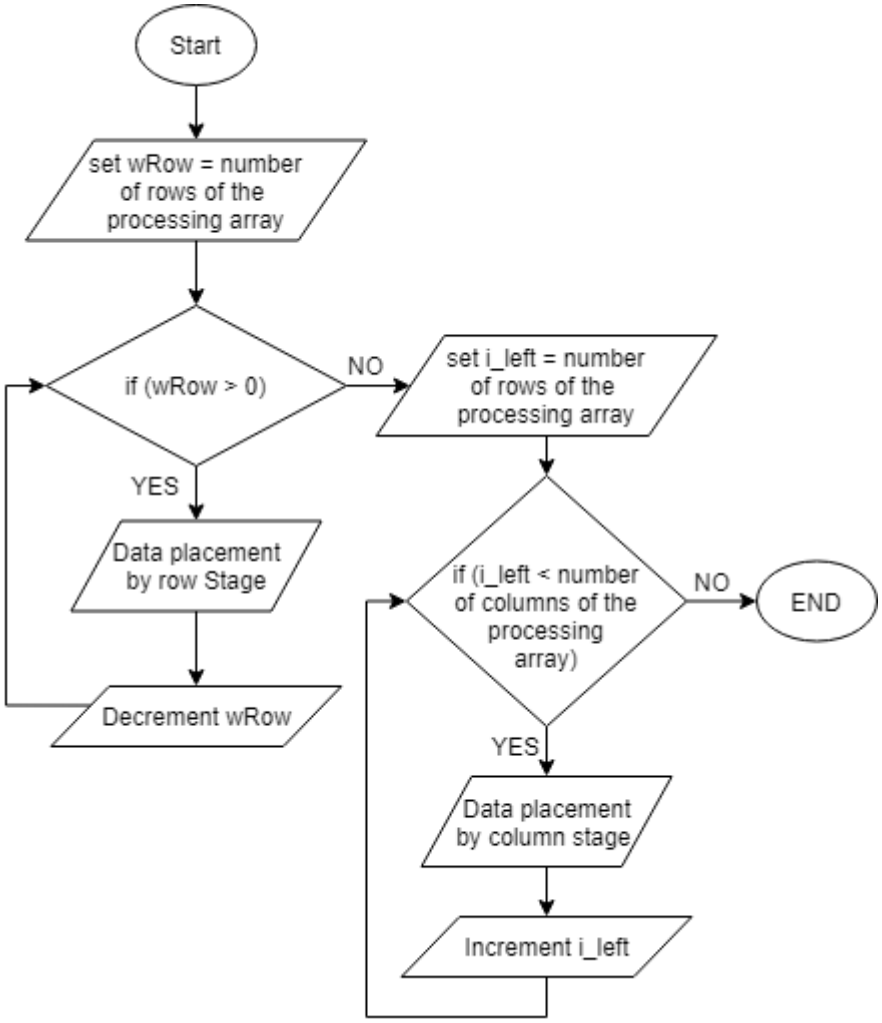


Figure 3-19 Data placement stage flowchart

The below flowchart represents Data placement stage by row.

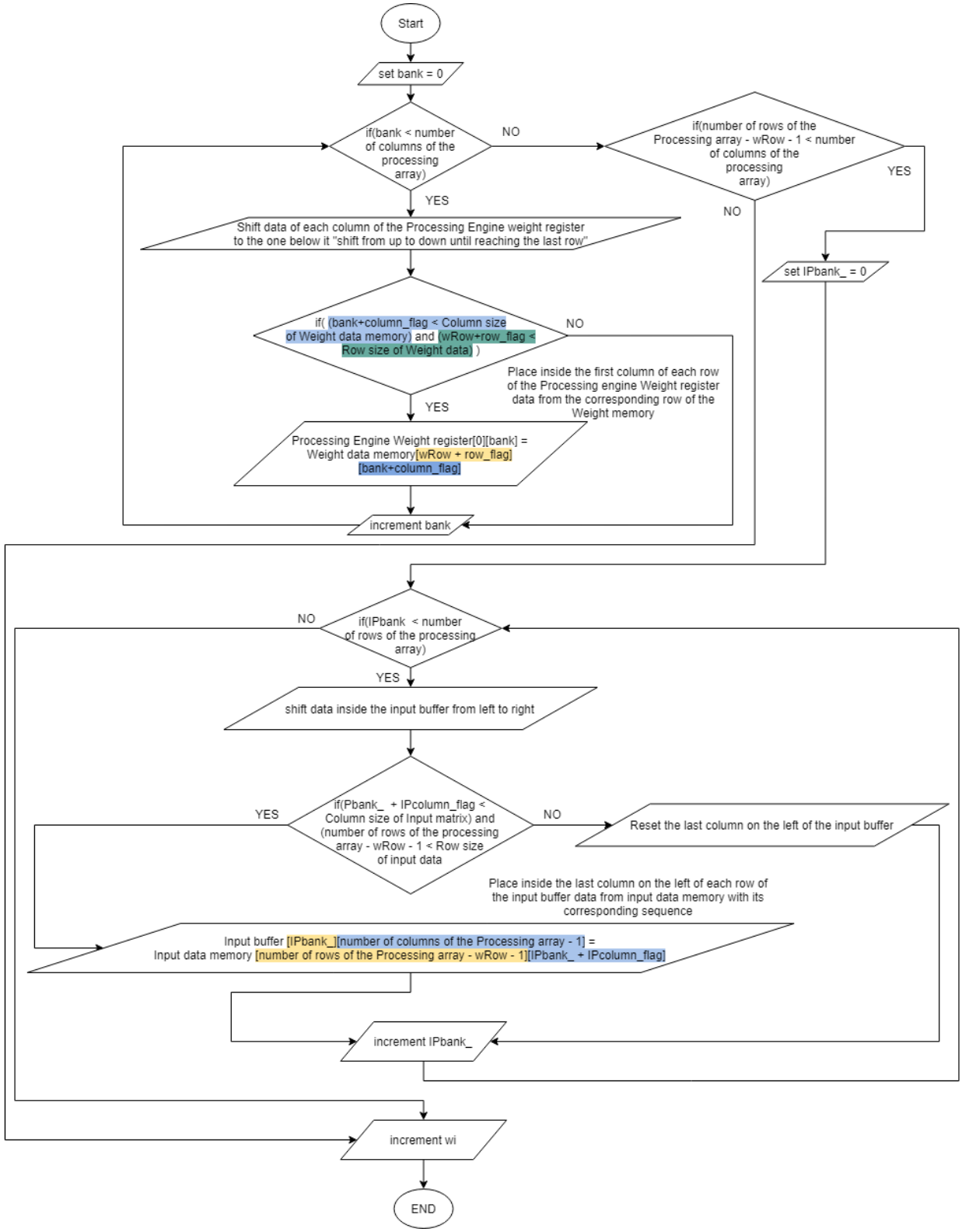


Figure 3-20 Data placement stage by row flowchart

The below flowchart represents Data placement stage by column.

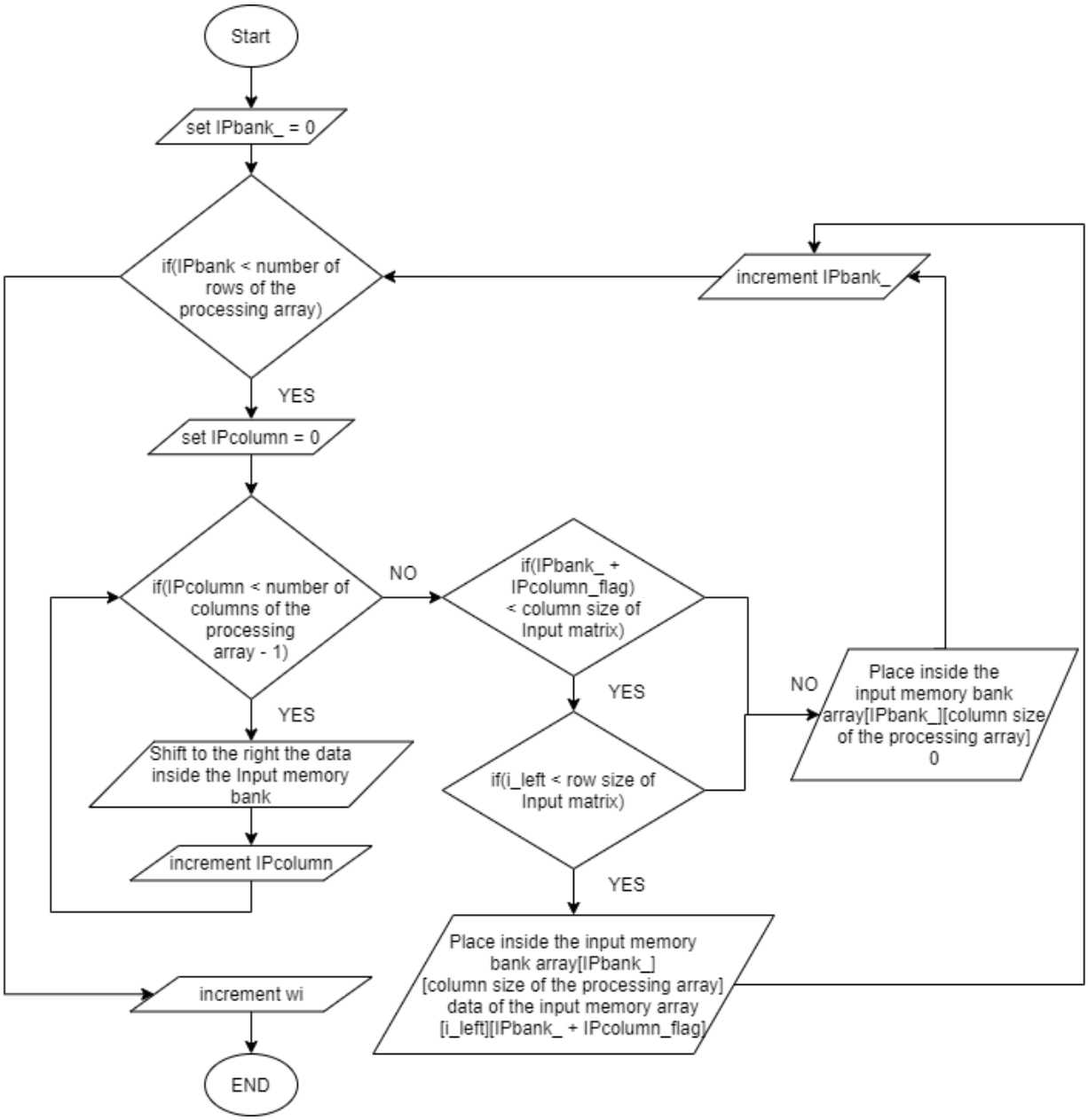


Figure 3-21 Data placement stage by column flowchart

Data processing stage will deal with the weights inside the Processing array after being organized in the previous stage. In addition to, shifting all input data allocated inside the input buffer, and continuously receiving only input data from the main memory as in Data placement stage. This stage contains two smaller stages, first stage deals with the upper part of the

Processing Array by receiving input data inside the processing element input registers and shifting them all along the Processing array with multiplying and adding the data inside the Processing array elements. The last stage deals with the lower part of the Processing Array by receiving the accumulated data from the last row of the upper part of the processing array to the accumulation part, and adding them with the data from the previous general cycle if they were from the same input column, as explained in the previous sub-chapter and then, send them back to the main memory.

The below flowchart represents the Data processing stage.

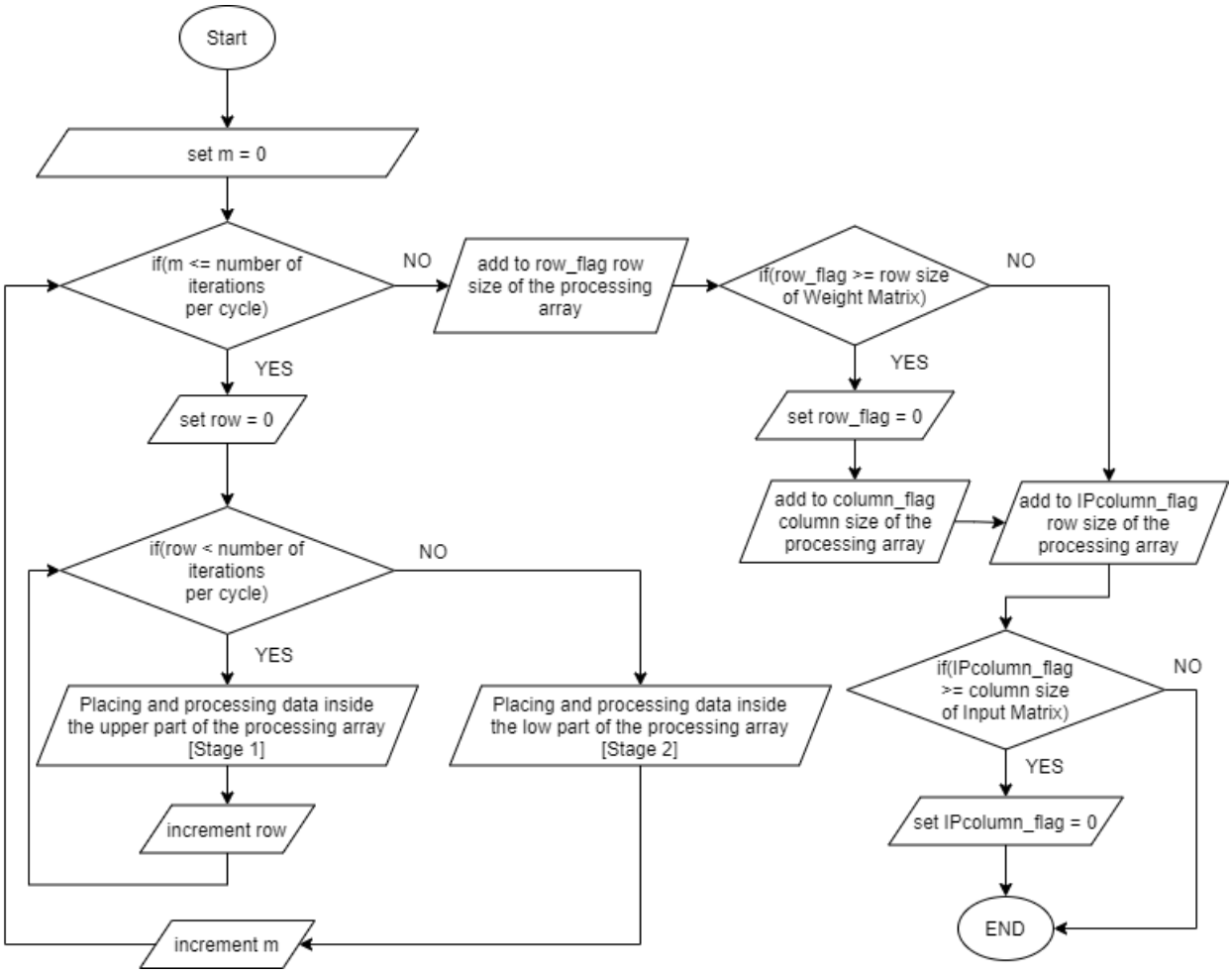


Figure 3-22 Data processing stage flowchart

The below flowchart represents the Data processing stage of the upper part.

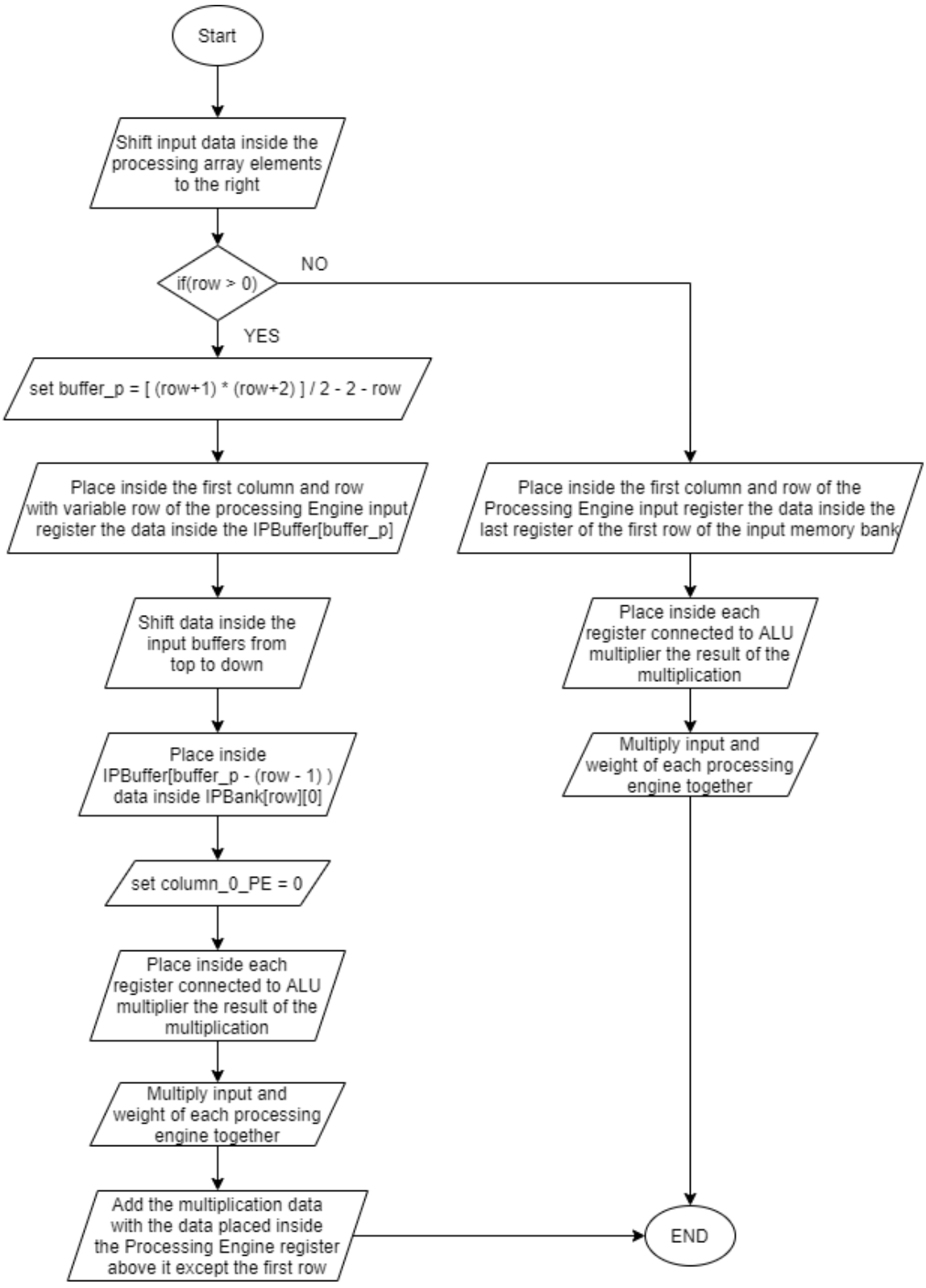


Figure 3-23 Data processing stage of the upper part

The below flowchart represents the Data processing stage of the lower part.

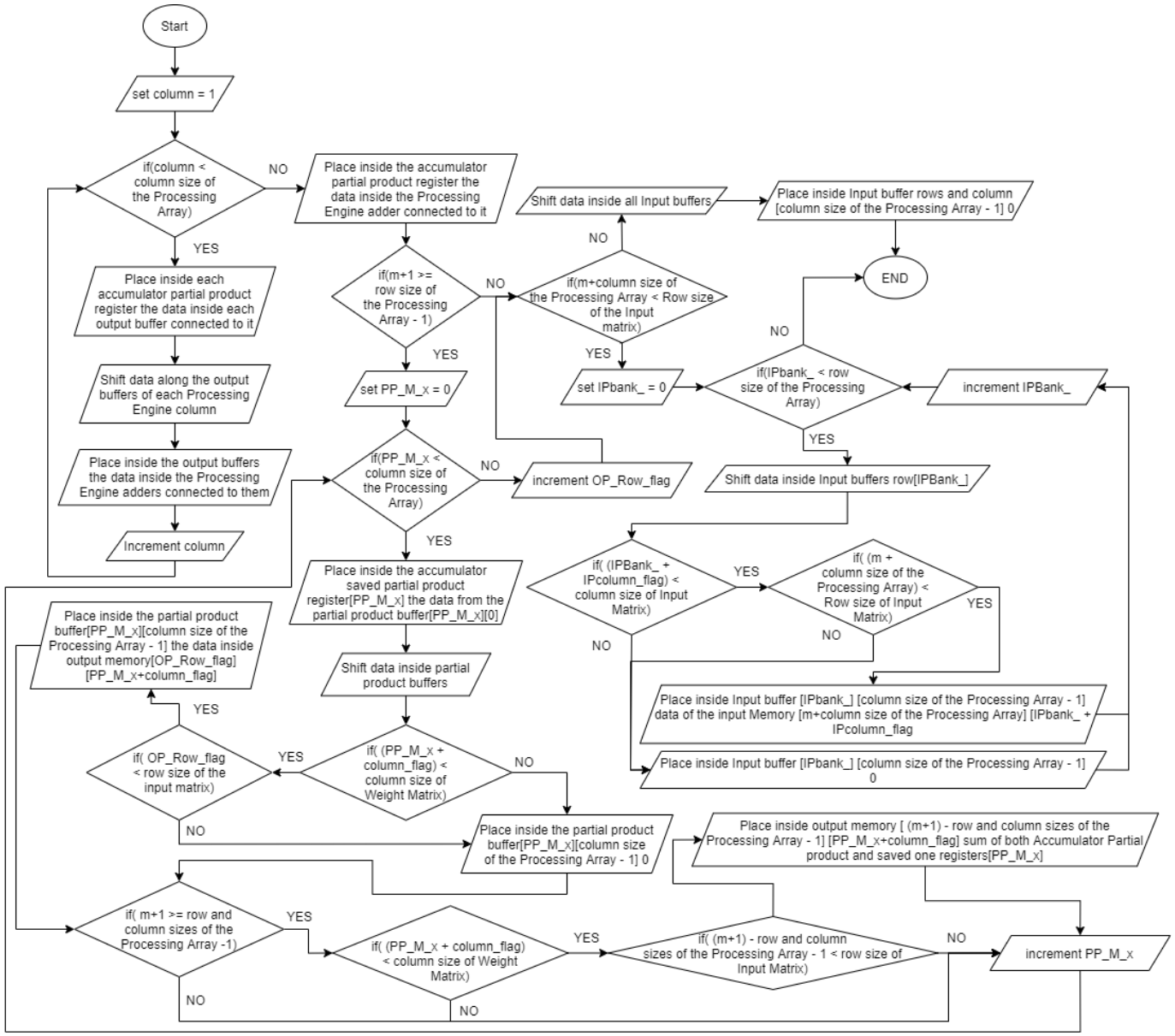


Figure 3-24 Data processing stage of the lower part

Finally, resetting all the processing array registers, which means, in our case, placing zero's inside Processing Array registers.

3.6 WORK CONTRIBUTION

As explained before how data flow through this design, and how can they be processed. It is possible now to discuss, why can this work give better results from energy, area, and latency points of view. Since the amount of accesses required by this design from and to the processing array and main memory is very limited, this means, that the power that is required is very low comparing to the design of *GPU*s that usually consumes tens if not hundreds of watts per cycle. While from area point of view, most *GPU*s are power hunger and contains millions of *ALU*s, while this design it is parametric that means, the number of elements can be tuned, to satisfy the desired neural network, and in most cases, hundreds or barely thousands of *ALU*s are pretty sufficient to run most of nowadays neural networks. Lastly, from latency point of view, since those *ALU*s are connected to each other's, bypassing of data is much faster in low-scale design of this work. It is important to hint out, that in some cases, if the design is very large scale, the latency will increase, and *GPU*s may give better results in such scenario.

In the upcoming chapter, experiments will be held out and show results of how it was used and the quality of the design, with their results.

Chapter four

4 EXPERIMENTS AND RESULTS

In this chapter, the reports of the design and the experiments made on it, will be shown, and analyzed.

The aim of this chapter is to test several real benchmarks on the proposed accelerator and show how can the dimensions of a convolution layer, can differ from another one, if applied on this proposed accelerator. Also, analyze the effectiveness of the accelerator on these convolution layers if it gets bigger.

The accelerator dimension can certainly differ with a positive or negative effect, based on the convolution it is tested on. That is why, three different dimensions will be proposed in this experiment.

4.1 SYNTHESIS AND OPTIMIZATION

It is possible to start synthesizing these three accelerators now, which will be referred to it later on by configuration, as the accelerator design is the same, while the dimensions of the systolic array will just differ. However, the three dimensions of the accelerator proposed are:

[1] 8 x 8 [2] 16 x 16 [3] 32 x 32

These three configurations will have different characteristics, since their hardware resources are not the same anymore. In order to know these aspects, we can use *Synopsys* tools.

4.1.1 DESIGN SYNTHESIS

Since the exported *Verilog* netlist only represents a functional architecture, it is time to transform this functional architecture into physical architecture.

4.1.1.1 DESIGN COMPILER

It is one the products offered by *Synopsys*, that is used in this work in order to synthesize a functional netlist using a technology library to have a physical architecture, with the characteristics of the selected technological library, then the ability analyze and elaborate this design with many aspects that are required in order to physically implement this design. Also, it is also possible to create virtual clocks assigned to the clocks of the design, in order to check for suitable frequency adopted to the design.

4.1.1.1.1 DESIGN-WARE

In addition to the features offered by the design compiler. Another feature it has, which is, a library where it has a collection of Arithmetic components and much more. This helps the design compiler to choose from this library the best combination of components that can produce the best trade-off between area and frequency.

The selection of the combinations occurs during the compilation process, and after it ends, the design characteristics can be known, such as, its area, and if the virtual clocks assigned were violated or not. Also, power analysis has been performed using *Prime time*,

4.1.1.2 PRIME TIME SHELL

This is one of the most critical steps in the design flow, since it verifies whether, this design would not violate any of the timing constraints performing static timing analysis. In addition to that, it performs an estimation of the power consumption.

After synthesizing the Verilog netlist, and having a post-synthesis netlist, made up of gates, and having *Standard delay format* (*SDF*) which has all the timing data of the design, it is possible to check the total dynamic power of the cells, by collecting the signal probabilities through a gate-level simulation. During the gate-level simulation random values has been used for the weights and the input activations.

This data collection is saved inside the *VCD* file. By reading the *VCD* file, post-synthesis netlist and the *SDF* file, a detailed estimation of the dynamic power consumption of the design can be generated.

4.1.2 AREA ANALYSIS

As explained in the previous chapter, the importance of Design compiler tool. Firstly, there was a script prepared in order to synthesize these three configurations in a quick way, since the constraints applied on all of them were exactly the same. The script was made up of some commands, to be run on the Design compiler, and it is written in *Tool command language* (*TCL*).

Design the area and maximum frequency reports of each configuration, were extracted. However, it was obvious, that the area of the biggest processing engine array configuration, will result the biggest area.

	8 x 8	16 x 16	32 x 32
Area [um^2]	1350.795	5415.895	21916.134

Table 4-1 Area analysis of multiple configurations of Processing Array

It is found that, by doubling the processing engine array, the area increases quadratically, and this is conceptually true. In a sense that, by doubling the array, means, both array row and array column are doubled, so, it is more like, cloning the resources double of times, in two side, which is quadratically related. This was the first verification, in order to check, that the high-level HDL conversion to Verilog is correct.

As this design take into consideration, that data flow should be correctly ordered, so that the data do not lose its sequence, lots of registers were internally placed in the processing engine as explained in the previous chapter. Yet, the combination cells found in the design, were much more than sequential cells. This is why, the arithmetic logic unit, placed in each processing element, contains an adder of 64 bits and also a multiplier, which generally, in low level design, they

breakdown into multiple cells, unlike, the registers, which are considered as a single cell for each bit.

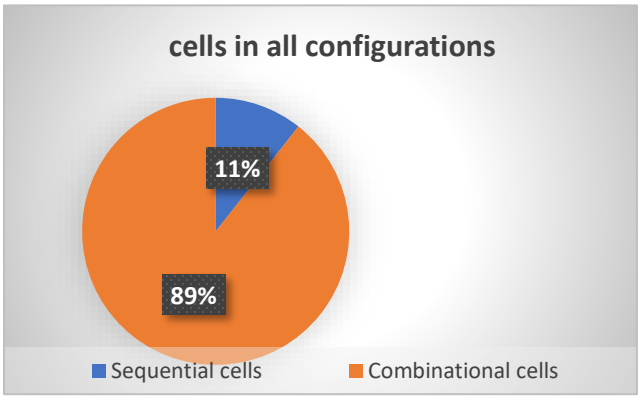


Figure 4-1 Sequential vs Combinational cells area analysis in all configuration

Clearly, the reports state that, most of the area is occupied by combinational cells. The below equation can be used, in order to estimate the conceptual number of sequential cells, in this design.

$$\left((PA_{Row} * PA_{Column} * 6) + {PA_{Column}}^2 - 2\right) * nbits$$

Equation 4-1 Number of sequential cells estimation

Since the design compiler, optimizes the internal registers of the processing array, the above equation, is considered to find, the maximum number of combinational cells. The total area found in Table 4-1, was calculated in design compiler by using technological libraries provided by *STMicroelectronics* of technology with unit area of 65 nm, called *STcmos65*.

The below equation can be used, in order to estimate the conceptual number of combinational cells, in this design.

$$PA_{Row} * PA_{Column} * nbits * 50$$

Equation 4-2 Number of combinational cells estimation

Certainly, as the above equation is considered to find, number of combinational cells, it is not very accurate. In a sense that, in low level design, the design compiler chooses from the design ware,

the best combination of cells of different architecture of adders and multipliers. Since the clock was fixed to 7.4 ns, the design compiler will rely more on area, and choose parallel architectures, allowing the clock to be the same, and increase the combinational cells. However, Equation 4-2, was concluded from compiling multiple configurations and resulting an average coefficient which is 50.

4.1.3 POWER ANALYSIS

As explained in the previous chapter, prime-time tool was used to estimate the power, by importing the *VCD*, *SDC*, and netlist files. The below table shows power estimated of all the test configurations.

	8 x 8	16 x 16	32 x 32
Power [mW]	0.0264	0.0929	0.3552

Table 4-2 Power analysis of the Processing Array in multiple configuration

Right like area analysis, power is increasing, approximately quadratically since area is increasing with almost the same relation. The reason behind why power is not increasing quadratically as much as area is that the technology library used, provides multiple levels of threshold voltages. Allowing the cells that are critical, to use low level threshold, so it does not violate latency, but leads to have more leakage power. This is why the ratio of the 32x32 configuration to 16x16 is higher than 16x16 to 8x8, since more cells in the 32x32 configuration will be placed in the critical paths, so these gates, will use low level threshold voltage.

4.2 LATENCY ANALYSIS

As explained in the previous chapter, how was the maximum frequency of the smallest configuration obtained, then it was used among all the other configurations, so same clock frequency will be the same for all the other configurations. However, as the purpose of this work,

to test this accelerator using different configuration on multiple actual benchmarks, to simulate the real latency. So, 10 different benchmarks, from different neural networks, and specific layers,

	Cnn6FER		***MobileNetV1***		***MobileNetV2***		***ResNet9***		***ResNet34***	
	conv2d L5	conv2d L19	conv2d L1	conv2d L56	conv2d L15	conv2d L99	conv2d L1	conv2d L10	conv2d L6	conv2d L84
Input shape	32,48,48	128,12,12	3,224,224	1024,1,1,1	114,56,56	1280,1,1	3,24,24	128,112,112	64,56,56	512,7,7
Output shape	32,48,48	128,12,12	32,112,112	1001,1,1,1	24,56,56	1001,1,1	32,112,112	256,112,112	64,56,56	512,7,7
Kernel shape	32,32,3,3	128,128,3,3	32,3,3,3	1001,1024,1,1	24,114,1,1	1001,1280,1,1	32,3,3,3	256,128,3,3	64,64,3,3	512,512,7,7
Matrix A (x, y)	2304, 288	144, 1152	12544, 27	1, 1024	3136, 114	1, 1280	50176, 27	12544, 1152	3136, 576	49, 4608
Matrix B (x, y)	288, 32	1152, 128	27, 32	1024, 1001	114, 24	1280,1001	27, 64	1152,256	576, 64	4608, 512
Memory accesses [$*10^6$]	21.233664	21.233664	10.838016	1.025024	8.580096	1.28128	86.704128	3699.376128	115.605504	115.605504
Latency [ms] with conf. 8x8	2.4796512	2.8472832	1.4879328	2.8643328	1.051947	3.580416	11.8871232	428.5246464	13.464921	19.6411392
Latency [ms] with conf. 16x16	0.6263064	0.8141184	0.3726936	1.4321664	0.3768672	1.790208	2.9732016	107.3357568	3.3918048	6.5470464
Latency [ms] with conf. 32x32	0.1597734	0.2546784	0.0935286	0.7274496	0.0956376	0.909312	0.7440108	26.9362368	0.8607384	2.4551424

Table 4-3 10 different benchmarks tested on the proposed configuration

where taken to test on. The selection of these benchmarks was based on, different characterization of these benchmarks, as shown in the table below. From Table 4-3, the selected benchworks,

results in different matrices (A, B), which were created using *GeMM* (General Matrix to Matrix Multiplication), by using *im2col*. Furthermore, these differences lead to quiet different latencies, eventually. However, in order to show the gain increased by increasing the processing array, latency of configuration 8x8 which is always having the biggest latency was normalized to 1, and that normalized the rest of the configurations, as show in the Figure 4-2.

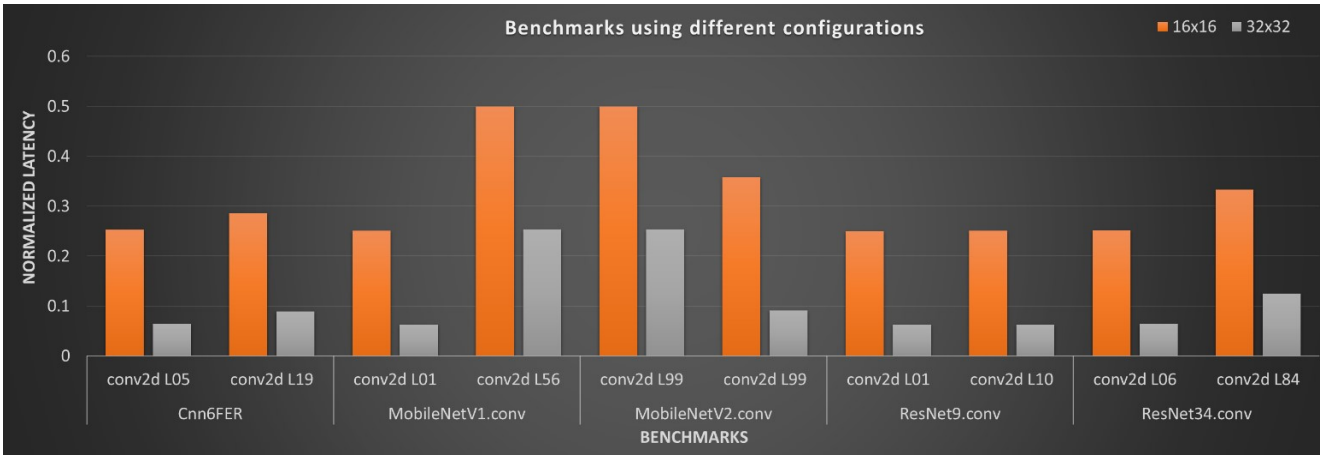


Figure 4-2 Normalized latency with respect to 8x8 configuration on the proposed benchmarks

It is clear from Table 4-1 that, the bigger the matrices are, the bigger the latency. Taking into example, *ResNet9* neural network, and its convolution layer, number 10, the latency of the configuration of processing array [8x8] was found approximately *428.5246464 ms*, this is due to the big size of matrices A and B. By using the below equation, it is possible to calculate the latency in milliseconds, by knowing the parameters of matrices A and B and the frequency of the design.

$$Ceil\left(\frac{Matrix_{B_{Row}}}{PA_{Row}}\right) * Ceil\left(\frac{Matrix_{B_{Column}}}{PA_{Column}}\right) * \left((PA_{Row} + PA_{Column}) + Matrix_{A_{Row}} + Max(PA_{Row}, PA_{Column}) - 1\right) * \left(\frac{1}{frequency_{kHz}}\right)$$

Equation 4-3 Latency calculation in milliseconds

It is certain now, that, the bigger the dimensions of the matrices, the more gain achieved. Taking the previous benchmark as an example, which has the biggest latency among all the selected benchmarks, yet, in Figure 4-2, it is having the least values. Also, in some benchmarks, it seems that by doubling the configuration, the latency decreases quadratically, while others not. Taking *Cnn6FER* neural network, layer number 19 and *MobileNetV1* neural network, layer number 56 as example, have *2.8472832* and *2.8643328* milliseconds, respectively, in configuration 8x8, while in configuration 16x16, changed to *0.8141184* and *1.4321664* milliseconds, respectively. Noticing, how close the latencies were in configuration 8x8, and diverged away in bigger configurations. This is due to the layer shape, which was in this case L19 input matrix is consisted of 144x1152 dimension, while the weight matrix here is 1152x128, which defines it has a compatible weight shape with the highest configuration, this means that all the 32x32 PEs will be filled out with

weights and not resource will be wasted out. While layer 56 has input matrix of 1x1024 and weight matrix of 1024x1001, this means that the IPs that will be processed is very low considering that there will be no streaming of IPs will be required in addition to that, the weights columns is not compatible, that means lots of resources will be wasted out.

4.3 MEMORY ACCESSES ANALYSIS

In this architecture, there are 3 access ports, that are connected to the main memory, as explained in the previous chapter. In this sub-chapter, Table 4-3 data is used. There are some benchmarks that have the same number of memory accesses, but, yet, different latencies, this means that this benchmark is more optimized than the other one. For instance, *Cnn6FER* neural network, convolution layers number 5 and 19, both having *21.233664* * 10^6 accesses, while *2.4796512* and *2.8472832* milliseconds, respectively, as latency. On the other side, by comparing *Cnn6FER* neural network, layer number 19 and *MobileNetV1* neural network, layer number 56, both benchmarks, have almost same latency in the 8x8 configuration, while number of memory accesses in both are *21.233664* and *1.025024* * 10^6 accesses. The reason behind this phenomena is mainly behind the reusing of the input matrix, for instance, the best scenario would be low number of columns for the input matrix, since no matter the number of rows of the input matrix, they will all be streamed in a single general cycle, yet, when having huge weight matrix, especially large number of columns, it will be streamed every general cycle, even though this is extensive, yet, this is the most reusing of inputs can be used. Certainly, this explains out all the scenarios behind why the convolution shape can differ in the latencies and memory accesses. However, more examples are shown below.

	ResNet34											
	conv2d L37				conv2d L68				conv2d L74			
Input shape:	128	28	28		256	14	14		512	7	7	
Output shape:	128	28	28		256	14	14		512	7	7	
Kernel shape:	128	128	3	3	256	256	3	3	512	512	3	3
Matrix A Row:	784				196				49			
Matrix A Column:	1152				2304				4608			
Matrix B Row:	1152				2304				4608			
Matrix B Column:	128				256				512			
Memory accesses x 10^6	115.605504				115.605504				115.605504			

Latency_with_configuration_8x8	13.759027	ms	14.93545	ms	19.641139	ms
Latency_with_configuration_16x16	3.5420544	ms	4.1430528	ms	6.5470464	ms
Latency_with_configuration_32x32	0.9366624	ms	1.2403584	ms	2.4551424	ms

Table 4-4 Benchmarks having 115.6 * 10^6 memory accesses comparison

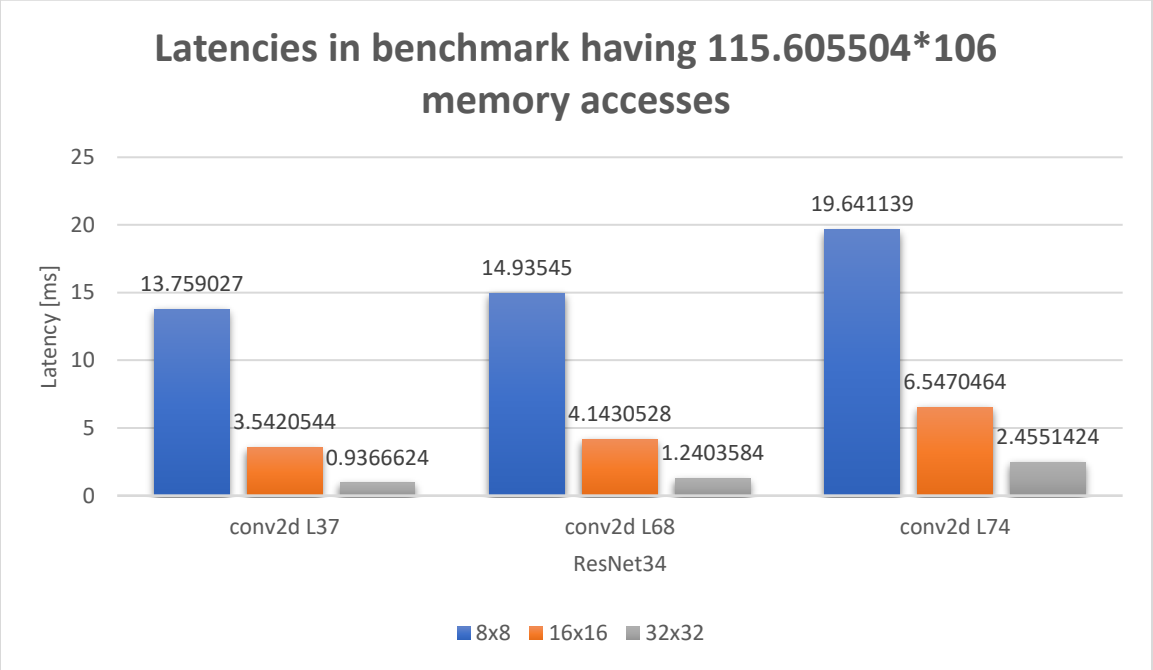


Figure 4-3 Latencies in benchmarks having 115.6 * 10^6 memory accesses

	ResNet34							
	conv2d L41				conv2d L72			
Input shape:	128	28	28		256	14	14	
Output shape:	256	14	14		512	7	7	
Kernel shape:	256	128	3	3	512	256	3	3
Matrix A Row:	196				49			
Matrix A Column:	1152				2304			
Matrix B Row:	1152				2304			
Matrix B Column:	256				512			
Memory accesses x 10^6	57.802752				57.802752			
Latency_with_configuration_8x8	7.4677248		ms		9.8205696		ms	
Latency_with_configuration_16x16	2.0715264		ms		3.2735232		ms	
Latency_with_configuration_32x32	0.6201792		ms		1.2275712		ms	

Table 4-5 benchmarks having 57.8 * 10^6 memory accesses

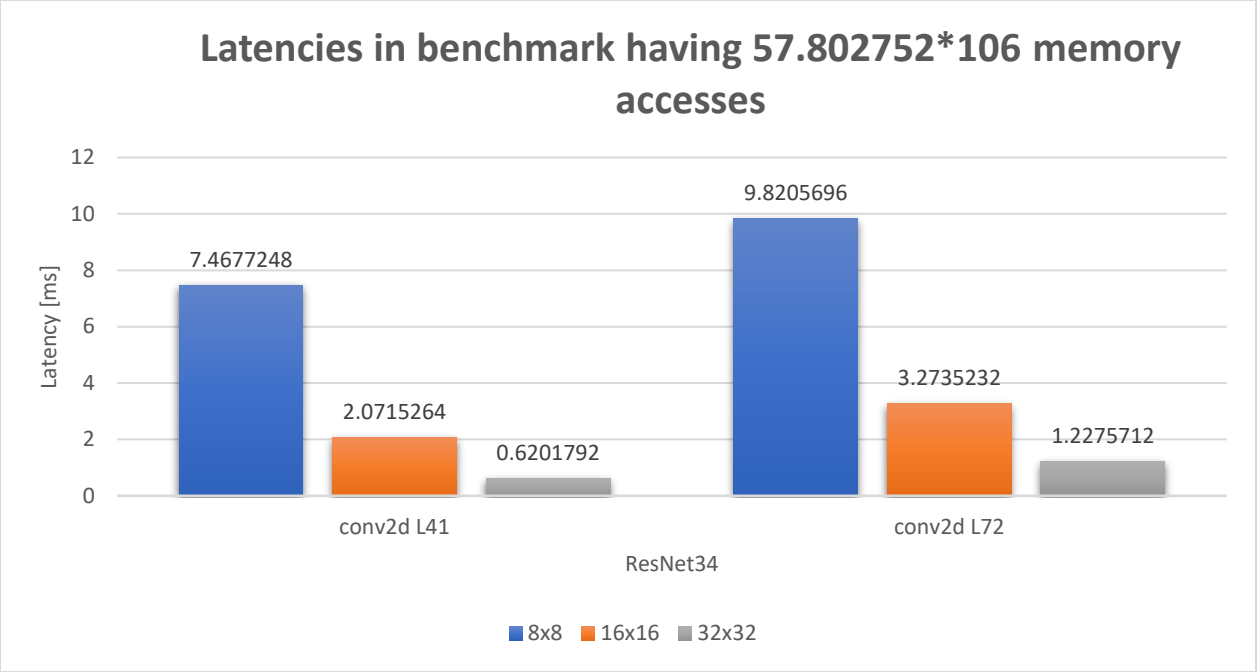


Figure 4-4 Latencies in benchmarks having 57.8 * 10^6 memory accesses

The below equation can be used in order to calculate the memory accesses.

$$Matrix_{A_{Row}} * Matrix_{A_{Column}} * Matrix_{B_{Column}}$$

Equation 4-4 Memory accesses calculation

4.4 DATA ANALYSIS

Using data in Table 4-1, Table 4-2, and Table 4-3, the table below is shown which refers to the 10 benchmarks selected is created.

	Maximum Latency	Average Latency	Minimum Latency	Power	Area
8x8	428.5246464	48.78293934	1.051947	26.4	1350795.155
16x16	107.3357568	12.56601696	0.3726936	92.9	5415895.657
32x32	26.9362368	3.3236508	0.0935286	355.2	21916134.13

Table 4-6 Area, Power, and Maximum, Average, and minimum latencies comparison in different benchmarks

5 CONCLUSION

The aim in this project was to create an integrated framework in Chisel for designing, verifying, and assessing the performance of Neural Network Accelerators based on Systolic array for processing unit for neural networks to be able to process it with relatively better performance. The approach used in this work was based on spatial architecture, which gives the ability for data to be more dynamic across the ALUs, unlike temporal architecture, which pushes data back and forth ALUs and main memory. The way data was flowing was based on systolic array strategy, which gives the Processing unit to have one of the three strategies:

- Input Stationary
- Weight Stationary
- Output Stationary

The strategy that was successfully used here was Weight Stationary, that allowed to place set of Weights inside the Processing Array and use the maximum possible number of inputs with their relevant Weight. Since neural networks does not come in shapes of inputs and weights, but input tensor and kernels, which comes in different kind dimension, it was required to use a strategy to convert the neural network parameters into a matrix multiplication in which it can be processed in this processing unit, this strategy is called *im2col*. However, this processing unit was designed by using a new high-level of HDL called *Chisel* that gave this work the power of designing this complex design in a superb optimized version and using powerful function that facilitated the description of this design. Not to mention, the possibility of testing and verifying that this design will give correct results by using a *SCALA* script that pokes and expects data to and from the HDL code, which resulted in a very clean environment having the possibility to trace and simulate everything happening inside the HDL code. Furthermore to that, the possibility of exporting an optimized *Verilog* code version, which has been used for synthesizing. Moreover, this design is parametric, which means, it is possible to control the set of the Processing Engines inside the Processing Array, in addition to, it is generic for all neural networks, which can only by simulated by placing it parameters inside the *SCALA* script and it will be converted to matrix multiplication

and calculate the number of cycles required for this neural network to be processed on the proposed design configuration.

5.1 FUTURE WORK

It is possible to improve this work by designing the remaining types of the systolic array, which are input and output stationaries, and simply create a script to check out which of those types will best fit the neural network that will be accelerated.

From the previous chapters, it is certain that the parameters of the neural network will manipulate the latency. For instance, possible future works can exploit the current framework to perform a design-space exploration where the dataflow adopted depends on the specific characteristics of each layer, for example, after converting the neural network parameters into matrix multiplication, if the input matrix is enormous, in such case, it will be much better to use the input stationary strategy than using weight stationary one.

Last thing can be done, is finding a tuning algorithm, that can find the best configuration.

REFERENCES

[1] *Joseph Risi, Amit Sharma, Rohan Shah, Matthew Connelly* and *Duncan J. Watts*, "Predicting history", in Nature Human Behaviour, 906-912, 2019.

[2] Codeburst, "What is the future of machine learning?", [Online]. Available: codeburst.io/what-is-the-future-of-machine-learning-f93749833645

[3] Potentia Analytics, "What is machine learning: definition, types, applications and examples", [Online]. Available: potentiaco.com/what-is-machine-learning-definition-types-applications-and-examples/

[4] Abeyon, "How do machines learn?", [Online]. Available: abeyon.com/how-do-machines-learn/

[5] IBM Cloud, "Machine Learning", [Online]. Available: ibm.com/cloud/learn/machine-learning

[6] Towards data science, "An introduction to reinforcement learning", [Online]. Available: towardsdatascience.com/an-introduction-to-reinforcement-learning-1e7825c60bbe

[7] SAS, "Neural Networks. What they are and why they matter", [Online]. Available: sas.com/en_us/insights/analytics/neural-networks.html

[8] TensorFlow playground, "What is a Neural Network?", [Online]. Available: playground.tensorflow.org/

[9] Marktechpost, "Neural Networks: Advantages and applications", [Online]. Available: marktechpost.com/2019/04/18/introduction-to-neural-networks-advantages-and-applications/

[10] Nature, "Does AI have a hardware problem?", [Online]. Available: nature.com/articles/s41928-018-0068-2

[11] *Ruizhe Zhao, Wayne Luk, Xinyu Niu, Huifeng Shi* and *Haitao Wang*, "Hardware Acceleration for Machine Learning", in Computer Society Annual Symposium on VLSI, *IEEE*, 2017

[12] Dynamic optimization, "Deep learning", [Online]. Available: apmonitor.com/do/index.php/Main/DeepLearning

[13] Investopedia, "What is a neural network?", [Online]. Available: www.investopedia.com/terms/n/neuralnetwork.asp

[14] *Marvin Minsky* and *Seymour Papert*, "Perceptrons. An Introduction to Computational Geometry", in M.I.T. Press, Cambridge, 1969.

[15] Medium, "Brief history of Neural network", [Online]. Available: medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec

[16] *R.E. Uhrig*, "Introduction to artificial neural networks", in Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics, *IEEE*, 1995

[17] *Manish Mishra and Monika Srivastava*, "A View of Artificial Neural Network", Advances in Engineering & Technology Research (ICAETR - 2014), IEEE, 2014

[18] Sophos news, "Man vs machine: comparing artificial and biological neural networks", [Online]. Available: news.sophos.com/en-us/2017/09/21/man-vs-machine-comparing-artificial-and-biological-neural-networks/

[19] *F. Rosenblatt*, "The perceptron: a probabilistic model for information storage and organization in the brain.", Psychological Review, 65(6), 386–408.

[20] Neuroelectrics, "Artificial Neural Networks – The Rosenblatt Perceptron", [Online]. Available: neuroelectrics.com/blog/2016/08/02/artificial-neural-networks-the-rosenblatt-perceptron/

[21] *K. Jahr, R. Schlich, K. Dragos* and *K. Smarsly*, "Decentralized autonomous fault detection in wireless structural health monitoring systems using structural response data", in proceedings of the International Conference on the Applications of Computer Science and Mathematics in Architecture and Civil Engineering. Weimar, Germany, 2015.

[22] The Asimov Institute, "The Neural Network Zoo", [Online]. Available: asimovinstitute.org/neural-network-zoo/

[23] Colah, "Understanding LSTM Networks", [Online]. Available: colah.github.io/posts/2015-08-Understanding-LSTMs/

[24] *Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang* and *Joel S. Emer*, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", in proceedings of the IEEE, Volume 105, Issue 12, 2017

[25] *Y. Chen, J. Emer* and *V. Sze*, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016

[26] *Moons B., Bankman D.* and *Verhelst M.*, "Hardware-Algorithm Co-optimizations", in Embedded Deep Learning, Springer, Cham, 2019.

[27] *N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit™", in Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on, IEEE, 2017, pp. 1–12.

[28] Google cloud, "TPU", [Online]. Available: cloud.google.com/tpu

[29] *Y.-H. Chen, T. Krishna, J. S. Emer* and *V. Sze*, "Eyeriss: An energy-efficient reconfigurable

accelerator for deep convolutional neural networks", IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127–138, 2017.

[30] Nvidia, "NVIDIA A100 Tensor Core GPU Architecture: UNPRECEDENTED ACCELERATION AT EVERY SCALE", [Online]. Available: nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[31] Microsoft, "Project Brainwave", [Online]. Available: microsoft.com/en-us/research/project/project-brainwave/

[32] Intel, "Intel® Stratix® 10 NX FPGA Technology Brief", [Online]. Available: intel.com/content/www/us/en/products/programmable/stratix-10-nx-technology-brief.html

[33] Greg Nash and Jim Moawad, "FPGAs For Deep Learning", in Argonne Training Program on Extreme Scale Computing, 2019

[34] Theano, "Convolution Arithmetic tutorial", [Online]. Available: deeplearning.net/software/theano/tutorial/conv_arithmetic.html

[35] *Manas Sahni*, "Anatomy of a High-Speed Convolution", [Online]. Available: sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/