Dynamic Reconfigurable Deep Neural Network Accelerator

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment
of the requirements for the degree

Master of Science

by

Pai Wang

May 2019

APPROVAL SHEET

This Thesis
is submitted in partial fulfillment of the requirements
for the degree of
Master of Science

Author Signature: 

This Thesis has been read and approved by the examining committee:

Advisor: Mircea R. Stan

Committee Member: Zongli Lin

Committee Member: Homa Alemzadeh

Committee Member: _____

Committee Member: _____

Committee Member: _____

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

May 2019

Acknowledgments

When I first arrived in UVa, I had no idea how to start my academic life in the engineering area. If I haven't received support from a lot of nice people, I would never achieve what I had done now.

I would like to thank my advisor first, Dr.Mircea R. Stan. He guided me to find what I am interested in and always offered me good suggestions whenever I met problems in my research. He helped me back on the right road when I got lost. The support coming from him is not only about the academic field but also about the wisdom to face difficulties in my life.

I would like to thank the HPLP lab, and my colleagues, Mateja Putic, Sergiu Mosanu, Vaibhav Verma and Tommy Tracy for their full support. I appreciate it that every morning when I arrived at the lab, I could see these familiar faces and work with them.

Contents

1 Introduction **1**

1.1 Contribution 1

1.2 Motivation 2

1.3 Organization 3

2 Background Information **4**

2.1 Neural Network 4

2.1.1 Convolutional Neural Network 5

2.1.2 Deep Neural Network 8

2.2 DNNs Accelerator 8

2.3 Systolic Array 9

2.3.1 2-D Systolic Array 11

2.3.2 Convolution With Systolic Array 11

2.4 Chisel 13

3 Related Work **14**

3.1 Weight Stationary 15

3.2 Output Stationary 16

3.3 No Local Reuse 17

3.4 Row Stationary 18

4 Reconfigurable DNN accelerator **20**

4.1 The Software Design-Krycek 20
4.1.1 Generate Configuration Candidates 20
4.1.2 Evaluation 23
4.1.3 Results 25
4.2 The Hardware Design-CNN-SAC 27
4.2.1 PE Design 27
4.2.2 SIMD 28
4.2.3 PE Array Design 30
4.3 Design In Chisel 33
4.3.1 Basic In Chisel 33
4.3.2 Systolic Array In Chisel 35
4.4 Dynamic Reconfigurable Systolic Array 38
4.4.1 Platform And Benthmark 38
4.4.2 Codes 39
4.5 Results 43

5 Conclusion **45**

6 Future Work **47**

List of Figures

2.1 Neural Network 5
2.2 Convolutional Neural Network 6
2.3 Input with Padding 6
2.4 Pooling Layer 7
2.5 Parameters of DNNs layer 8
2.6 von neumann Computer Architecture 10
2.7 1-D Systolic Array 10
2.8 2-D Systolic Array 11
2.9 Initial state 12
2.10 Mul.scala 13
2.11 Mul.v 13

3.1 Weight Stationary 15
3.2 5×5 Convolution In Weight Stationary 15
3.3 Output Stationary 16
3.4 No Local Reuse 17
3.5 TPU Architecture [11] 17
3.6 Eyeriss System Architecture [3] 18
3.7 Convolution With Stride = 1 18
3.8 Row Stationary 19

4.1 Configuration Generator 20

4.2 DNN Library 21
4.3 Xilinx Pynq Board 21
4.4 Xilinx AC701 Board 22
4.5 Systolic Array Class 22
4.6 Convolution Layer Class 22
4.7 Static VS Reconfiguratable 25
4.8 Energy Improvement 25
4.9 Performance Improvement 26
4.10 Basic PE 27
4.11 Inside Of PE 28
4.12 SIMD = 1 29
4.13 SIMD = 2 29
4.14 SIMD = 4 30
4.15 Base Architecture 30
4.16 SIMD = 4 31
4.17 SIMD = 8 31
4.18 SIMD = 1 31
4.19 SIMD = 2 31
4.20 PE Waveform 35
4.21 Systolic Array Tester 36
4.22 Systolic Array Waveform 37
4.23 Xilinx VCU110 Board 39
4.24 Diagonally Sum Up 42
4.25 Reconfiguration Energy Improvement 43
4.26 Reconfiguration Performance Improvement 43

List of Tables

2.1 Convolution Layers of VGG11 9
2.2 Results In PEs 12

4.1 ROW and SIMD 30
4.2 Convolution Layers of VGG11 38
4.3 Configurations Of Different PEs 39

Chapter 1

Introduction

1.1 Contribution

Different DNNs such as AlexNet [12], GoogleNet [19], VGG11 [17] and etc, have different data dimensions. When we map these Deep Neural Networks to the hardware part, the inputs with the changeable dimension will make hardware resource not highly used in some layers. In this thesis, we proposed a new Deep Neural Network Accelerator, which can specifically speed up the convolution process during the execution by dynamically changing the shape of PE array to make the hardware part as high as much used by inputs with different dimensions.

For specific hardware, such as an FPGA or an ASIC, it will have its computational units, which can handle matrix multiplications. Through using our architecture, we could change the configuration during DNNs computation process rather than keep all of the layers with the same configuration. But the question is how to decide which configuration list will be the most suitable for our network. So we put forward a simulator called 'Krycek', which will evaluate all configuration candidates and then filter the configurations above constraints that we put there. At the end of the Krycek, it will sort all rest of configuration candidates to find the best configuration choice for our DNNs.

When we have the architecture design and evaluation tool, we still need a verification tool that could check our model is good or not. In other words, we need to apply our DNNs to actual hardware and then run testbenches with the chosen configuration. So we proposed a scalable Chisel design called 'CNN-SAC'(Convolutional Neural Network In Systolic Array with Chisel), which has the ability to dynamically reconfigure our system according to the configuration generated by Krycek.

1.2 Motivation

Nowadays, machine learning is being used almost everywhere. When people go to the gym, they can calculate their Calorie cost based on how long they run or how many hours they work out. An agent in Stock exchange company could predict the price tendency of stocks with the help of analyzing previous stock's price. Also, doctors in the hospital may get more clues from the CT images by using machine learning in image processing. And the automobile, wearable devices and drones and other IoT devices, most of them are equipped with the power of machine learning. Through using deep learning, users have more chance to find the hidden characteristics behind the previous data. Machine learning now is not only the research topic but also changing our daily life.

Different users may apply machine learning algorithms in different areas. And they will pursue distinct metrics of deep learning, some of them could care about the accuracy because their fields are pretty sensitive to the error. Some of them may care about the performance for they need to get output as fast as possible. And energy efficiency is also an essential feature for low power device or other machines that want to cost less energy. The higher precision largely depends on the data itself and algorithms, which belong to the software part. But for performance and energy efficiency, if we have a board that is designed for Deep Neural Network rather than running own network on a generic processor like CPU and GPU, we may have more chance to achieve higher performance and lower energy cost.

A lot of specific boards designed for deep learning have already invented, called DNN accelerators or AI chip. These chip are kinds of ASICs, which have particular functionalities. Some of them are designed for image processing, and some of them target low-power wearable device. And some DNN accelerators are implemented on the automobile to make self-driving possible. In a word, under the desire of higher performance and lower cost of DNN execution, DNN accelerators have shown up and received a lot of success.

However, there is still space for us to exploit. When we dive into the deeper level of DNN accelerator design, the input's dimension of different DNNs varies mostly. And even for the same DNN but different layers, their input dimensions are distinct. To feed these diverse input data into the specific hardware with static configuration, it will require our board with the vast resource. So during the execution of layers that have low data dimension, our board could not be fully used, which will affect the performance and energy efficiency.

Also, we know that for Deep Neural Network computation, the majority of time is spent on convolution layers. So in this thesis, we mainly focus on solving the problem of how to make convolution layers execution on specific hardware with higher performance and lower energy cost.

Instead of using static configuration for every layer, we want to get a dynamically changeable board that can switch its shape to meet the different requirements of different inputs. Through using this dynamic configuration during DNN layer's execution, the hardware part

is trying to buffer the difference between input data's dimension and hardware resource's dimension. So dynamic reconfiguration exploits more space for us to improve the performance and energy cost versus only using static configuration when running the DNNs on hardware.

1.3 Organization

The following chapters describe background information, related work, design results and conclusion. In the background chapter, we introduce the terminologies and technologies that are used in this paper. The neural network section gives a brief introduction of what is neural network and where it can be used for. Also, this section talks a little bit about the machine learning algorithms which is going to be used in later chapters. And the section for DNNs accelerator in this chapter illustrates the relationship between deep learning and hardware. From this section, we clarify the basic definition of DNNs accelerator and explain the advantages of it. Then, the last two parts in this chapter, Systolic array and Chisel, one is the architecture used in this paper, and the other is the tool to map our neural networks in hardware.

The next chapter introduces some related research work, such as TPU [11],DianNao [4] and Eyeriss [3]. These researches have already achieved a lot of success in the DNN accelerators field. The ideas and strategies in these papers gave me a lot of inspirations for this thesis. This chapter begins with a series of strategies for DNN accelerators and then provides a brief description of each strategy's advantages and the DNN accelerator that use this strategy. After we have a whole picture of what we have and what we need, this chapter clarifies why we choose output stationary as our architecture design.

The chapter,'Reconfigurable DNN accelerator', is the key part of this paper. To make the design's idea and process clearly, we split this chapter into two part, Krycek and CNN-SAC, which represent the software part and hardware part separately. The Krycek part introduces how to generate configuration candidates, evaluate candidates and choose candidates from constraints. In the other hand, the CNN-SAC is the steps of mapping desired neural network to hardware board, which means how to get HDL(hardware description language) of our design. Also, in this part, we introduce the improvement of performance and power efficiency when the accelerators apply dynamic reconfiguration versus static configuration.

The end of this paper gives a summary of this whole research and clarifies the final results can meet with our motivation. Also, we admit there is still some work to be done in future research work.

Chapter 2

Background Information

2.1 Neural Network

For the neural network architecture, it has three layers normally: Input layer, hidden layer and Output layer. The main computation process happens in the hidden layer. The hidden layer is made of a series of neurons. Every neural has its weight, and the weight can be updated. So when the input data is passed to the hidden layer, the data will be multiplied with different weights in different neurons. Under this situation, every neuron may be connected with several input data like shown in Figure 2.1. Before the results could be passed to the output layer, all the data in every neuron will be summed up. For most of the machine algorithms, the weights in the hidden layer will be keep updated to minimize the value loss function. And in the output layer, the data will be fed into the activation function, which means that we want to keep an apart of the dataset. By doing the activation process, the complexity of dataset decreases largely, because an amount of data will be set as zero.

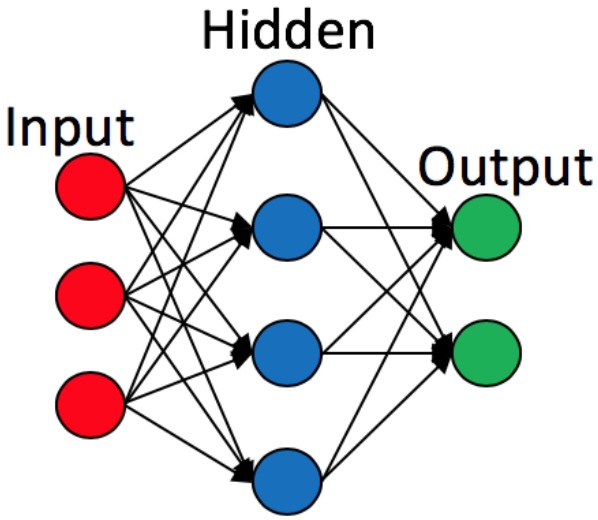

Figure 2.1: Neural Network

2.1.1 Convolutional Neural Network

The CNN(Convolutional Neural Network) was put forward by Kunihiko Fukushima [9]. This machine learning architecture is quite suitable for image processing. A lot of research work have already done,such as ImageNet [12],VGG [17],GoogleNet [20], etc. The main reason why CNN is suitable for image process or computer vision is CNN can extract the features of data and the relation between neighboring data. The basic architecture of CNN(convolutional neural network) is like Figure 2.2. In CNN algorithm, every data has its own dimension, which includes height, width and depth. And the weight kernel has the same depth as the input data. For example, in Figure 2.2, the first input data's dimension is 8x128x128, which means the depth of this input should be 8. So the according weight's depth should be 8. And the weight kernel will slide in input to do the dot multiplication with every part of input data by using a specific stride. When the kernel is moving from left to right, from top to bottom, the network is trying to enhance the features and find the relation of the data with its neighbor. This function is very important and useful to object detecting. For instance, when there is picture of a man, the convolution may try to get the profile of this person at first CNN layer. But for the next layer, it may draw the eyebrow, the lips... of this man.

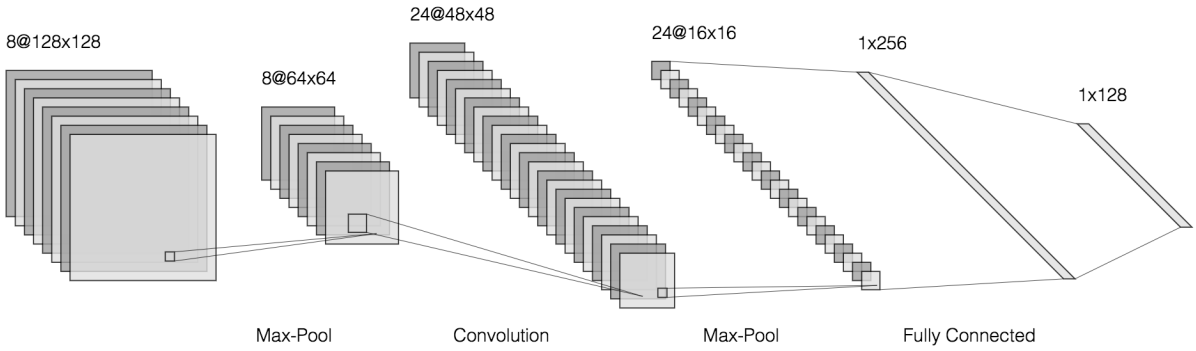

Figure 2.2: Convolutional Neural Network

Under this understanding, the result's after dot multiplication will decrease not only on the depth but also on the height and width. If the input is 32x32x3, and the kernel is 3x3x3, and the stride is 1, the output should be 30x30. But we don't want to shrink the height and width and depth in this process. So to keep the height and the width unchanged, we have to add some padding around the input data, like Figure 2.3. The padding is all 0s. Adding the padding will keep the output's height and width as the same as the input's. Also, because the padding is 0, which will not affect the results. But with the help paddings, the two dimensions of output data will not change after computations.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 2.3: Input with Padding

However, at the end of Convolutional Neural Network, it should be a list of possibilities. For example, we are going to using CNN to detect objects. After finishing the training process,

we could get a trained model. And when we feed new images to our trained model, it should give us the possibilities of objects, such as 0.5 chance is a car, 0.2 chance is a human...So we need to decrease the data dimension also. There will be two steps to shrink the data size. The first one is pooling layer, and the other one is a fully connected layer.

1	2	2	0	12	0	2
23	4	56	34	34	21	23
5	23	45	23	12	14	2
45	11	12	1	56	16	36
67	77	23	2	23	7	72
9	98	0	4	12	0	1
1	33	13	7	89	9	1

Figure 2.4: Pooling Layer

There are several strategies for pooling layer. The most popular one should be MaxPooling. Like in Figure 2.4, we use the max number to represent a part of data, such as the '23' should be used to represent '1', '23', '2' and '4'. Also, the size of the dark red part could be changed. Through this process, the result's dimension of convolution layer will decrease largely. Also, with the help of pooling layer, the complexity of computation will be reduced.

The second step is a fully connected layer. We could also regard fully connected layer as a kind of special convolution layer. The fully connected also has weight kernel, but the size of the kernel will be the same as the input data. For example, if the input of a fully connected layer is $7 \times 7 \times 512$, the kernel of this layer should also be $7 \times seven \times 512$. So the output size of the fully connected layer will contain a list, and the length of this list depends on how many kernels this layer has.

The previous paragraphs give a basic introduction of Convolution Neural Network. However, some details will not be discussed in this thesis such as activation function, cost function and softmax function, which are also extremely important for CNN. In this paper, we mainly focus on the convolution layer part.

2.1.2 Deep Neural Network

A deep neural network(DNN) is a neural network with multiple convolutional layers, pooling layers and fully connected layers. In this thesis, we regard DNNs as including a brunch of convolutional layers and pooling layers. From the previous section, we know that the width and height will be adjusted with the help of padding. But the depth will be changed to 1. However, the output of this layer will be the input layer of the next layer. Depth = 1 is not enough. If we increase the number of the kernel, the output will be the combination of depth=1, like Figure 2.5. The InW and InH is the width and height of input data. The OutD is the number of the kernel. The depth of the output layer depends on how many kernels it has. Also, a complete DNNs [12] [17] should normally have a convolutional layer, pooling layer and fully connected layer. The pooling layer is going to shrink the size of the dataset and the fully connected layer prepare the data for the end of the network, such as Softmax function.

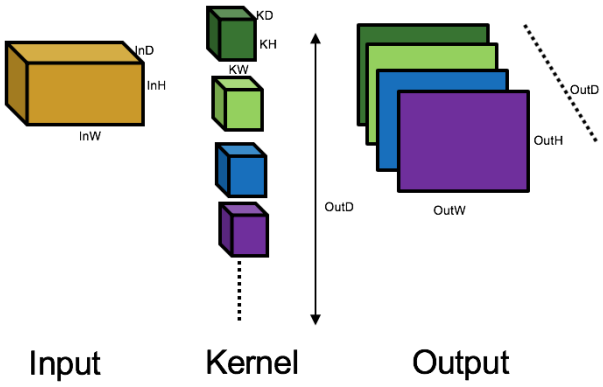

Figure 2.5: Parameters of DNNs layer

Also, in Figure2.5, the input is just one batch. But we can also handle several batches of input data, which means that you can deal with the number of images at the same time. Accordingly, you will get a bunch of output volume. The table 2.1 shows different data dimension in VGG11. The depth of each convolution layer changes dramatically. The first and second convolution layer's input data have smaller depth but larger height and width compared with other convolution layers. The last several convolution layers have the input data with smaller height and width but larger depth versus other layers.

2.2 DNNs Accelerator

The DNNs has achieved success in many areas such as facial detection [10], speech recognition [1], stock price prediction [21] and etc. Also, the DNNs has been used in tons of IoT devices,

DNN layer	Nin	Nout	Nij	Kij
CONV1	3	64	224 × 224	3 ×3
CONV2	64	128	112 × 112	3 ×3
CONV3_1	128	256	56 × 56	3 ×3
CONV3_2	256	256	56 × 56	3 ×3
CONV4_1	256	512	28 × 28	3 ×3
CONV4_2	512	512	28 × 28	3 ×3
CONV5_1	512	512	14 × 14	3 ×3
CONV5_2	512	512	14 × 14	3 ×3

Table 2.1: Convolution Layers of VGG11

such as AutoMobile [22]. The performance and power efficiency become important metrics we need to concern about besides accuracy. We want to find a strategy which has low power but better performance. This desire blossomed a lot of DNNs accelerators. These accelerators have the features to improve the energy-cost and performance of DNNs.

To map the specific the model to hardware, FPGA and ASIC will be the suitable platforms. Because compared with general purpose processor, FPGA or ASIC can meet the specific computational requirement of DNNs, which will be more possible to leverage the improvement of power and performance. Through minimizing the data movement, increasing bandwidth, throughput and data reuse [16] [3] [5] [11], these accelerators have already achieved great enhancement on energy-cost and performance.

For the DNNs model, they have training and interface process. The training process can be completed on software, and the interface process can be executed on hardware. Because the training process is very time-consuming and no necessary to run on hardware. When the model is already trained, the next step is to feed the input data and weight kernel to the hardware part at a kind of specific order. Different data arrangement will result in different metrics at end. In this paper, we don't care about the training part and will put most of the attention on the hardware part. It means that we train the model on software and then implement the trained model on our hardware.

2.3 Systolic Array

For the tradition computer architecture, shown in Figure 2.6, PE(processor element) takes the data from memory units and then continues operations. After PE finish its operations(Mul, Add, etc.), it needs to return the result back to memory units. So if the PE is busy, data come from memory cannot be fed into PE sequentially. However, in systolic array architecture, like in Figure 2.7, every PE will take a part of the task. When PE finishes its task, it will pass data to the next PE. So by using the systolic array, the input

bandwidth width could be increased largely. Also, the systolic array can achieve higher throughput compared with traditional architecture. And the third advantage of systolic array versus traditional computer architecture is its simplicity. A simpler architecture will be more friendly to be implemented on hardware.

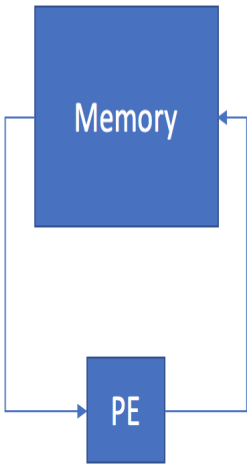

Figure 2.6: von neumann Computer Architecture

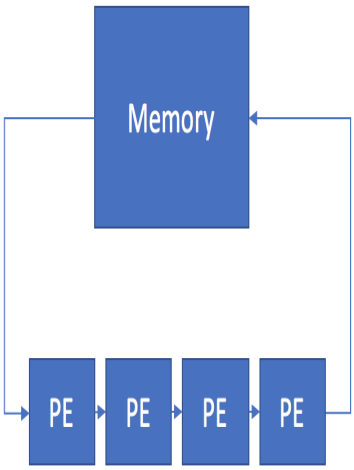

Figure 2.7: 1-D Systolic Array

2.3.1 2-D Systolic Array

The one dimension Systolic Array is much like the pipeline. Every PE finishes its task and passes data to the next PE. The result can be obtained at the end of the systolic array. In other words, it likes the assembly line, and every PE is an assembly work. So each PE will focus on its mission and then pass the item to next PE. And for the 2-D systolic array, its

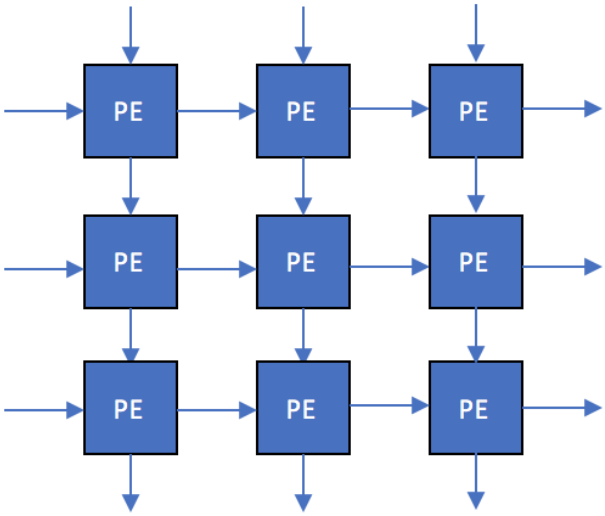

Figure 2.8: 2-D Systolic Array

data come from two dimensions. Like in Figure2.8, the PE will receive input data from the left side and top and then pass the data to the right side and bottom.

2.3.2 Convolution With Systolic Array

In this paper, we are going to do convolution by using 2-D systolic array on the hardware part. For example, the below Input, Weight and Output matrices, when the stride equals to 1 without padding,

$$Input = \begin{bmatrix} f00 & f01 \\ f10 & f11 \end{bmatrix}$$

$$Weight = \begin{bmatrix} w00 & w01 \\ w10 & w11 \end{bmatrix}$$

$$Output = Input \times Weight = \begin{bmatrix} f00w00 + f01w01 + f10w10 + f11w11 \end{bmatrix}$$

The most important feature of Systolic array is that every computational unit only takes a part of the whole task. As shown in Figure 2.9, the data from Input and Weight will be passed to the PE Array in the specific order. The w10 will arrives at PE2 one cycle later than w00 arriving at PE1. Also, the f10 and f01 will arrive at PE3 and PE1 separately but at the same time. The Weight and input like two rivers flow to the same plain. One comes from top to bottom, the other passes from left to right. When these two rivers meet with each other, they do computation and leave their results on the plain. Then they keep flowing at the origin direction. So in Figure 2.9, when w00 and f00 arrive at PE1, they will do multiplication.

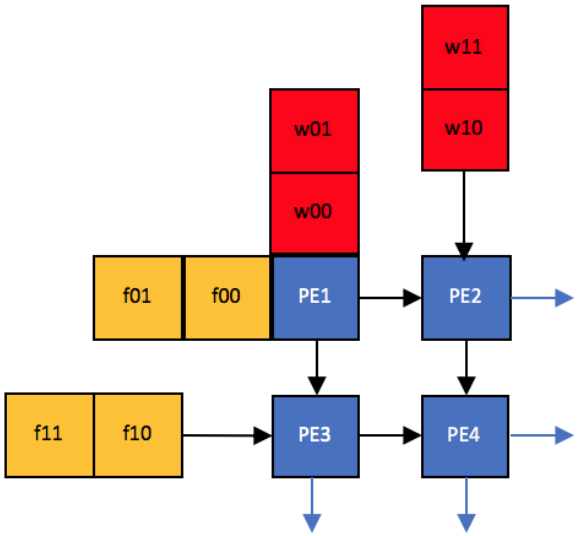

Figure 2.9: Initial state

Cycles	1	2	3	4
PE1	f00 × w00	f01 × w01		
PE2		f00× w10	f01× w11	
PE3		f10× w00	f11 × w01	
PE4			f10 × w10	f11× w11

Table 2.2: Results In PEs

The table2.2 shows the partial sums in different PEs at different cycles. At cycle 1, data from Input and weight only arrive at PE1, so the rest of the PE array will be idle. For the next cycle, not only PE1 will receive data, but also PE2 and PE3 will get data. At the last cycle, when PE4 finishes its operation, it's time to sum up the partial results in PE1 and PE4 to get the final result.

2.4 Chisel

Chisel is a new language designed for advanced digital circuit design [2]. As we have known, if you want to implement your model on hardware, you need to get RTL code first. Usually, we can write it on VHDL or Verilog. It's possible to get any models or architectures by using HDL(hardware description language) such as VHDL or Verilog. But in reality, it will be extremely difficult and time-consuming to get even a five layers DNN by using these two HDLs.

Chisel is based on Scala and much more like software coding. The standard compilation of Chisel to Verilog is converting Chisel file to Firrtl [13] file, which is user's Chisel RTL. And then this Firrtl(Flexible Internal Representation for RTL) file can be transformed to Verilog, which can be applied to FPGAs or ASICs.

```
import chisel3._
class Mul extends Module{
  val io = IO(new Bundle{
      val a = Input(UInt(8.W))
      val b = Input(UInt(8.W))
      val c = Output(UInt(16.W))
  })
  io.c := io.a * io.b
}
```

Figure 2.10: Mul.scala

```
module cmd13HelperMul( // @[:@3.2]
  input         clock, // @[:@4.4]
  input         reset, // @[:@5.4]
  input  [7:0]  io_a, // @[:@6.4]
  input  [7:0]  io_b, // @[:@6.4]
  output [15:0] io_c // @[:@6.4]
);
  wire [15:0] _T_11; // @[cmd13.sc 8:16:@8.4]
  assign _T_11 = io_a * io_b; // @[cmd13.sc 8:16:@8.4]
  assign io_c = _T_11;
endmodule
```

Figure 2.11: Mul.v

For example, if we want to build an 8 bits multiplier, like Figure 2.10, we need to declare Mul class first. When we are making this class, we are trying to design a module that can be used by other modules in this same package. Also, we need to specify bitwidth of each variable and then Mul.v could be generated through verilator [18]. Chisel also provides a testing feature that you can put your expected output in the test file to check whether generated output can meet with the expected output.

Chapter 3

Related Work

A lot of DNN accelerators have been invented to speed up the DNN process and enhance power efficiency at the same time. The common strategies applied by these accelerators are reducing data movement and changing memory access. In other words, these accelerators are to increase data reuse and shift to low-cost memory access path from high-cost memory access path. If the data passed to the computational array can be reused, the computation process will be shortened because the processor array will reduce the time for getting the data. Also, we know that accessing DRAM will cost more than getting data from SRAM. When the needed data are frequently passed from SRAM, the performance and energy metrics will be largely improved.

In the convolution layer of Deep Neural Network, there are three types of data we usually care about most, the input data such as pixels, the weight(kernel) data, and the output data. And for the convolution computation, a kernel matrix will slide the whole input matrix. Also, one input matrix can be used by a branch of different kernels. So both input data and kernel can be reused. For memory access, we know reading and writing data from DRAM will need more time and energy compared with SRAM. If we could switch the reading and writing process from DRAM to buffer or register as much as possible, the cost spent on memory access can be lowered down mostly.

Currently, several strategies are being used to resolve the memory access issue which can increase data reuse and change to use other low-cost memory location to read and write data. Through changing the data movement or the data location, the weight stationary, output stationary, no local reuse, and row stationary have achieved higher performance and lower energy cost compared with traditional DNN accelerators architecture.

3.1 Weight Stationary

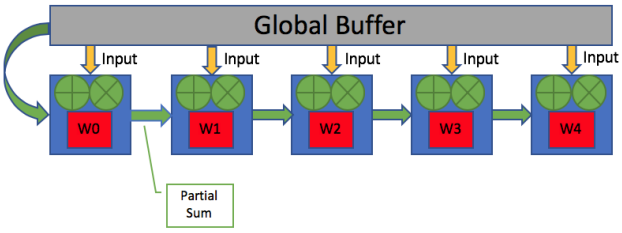

Figure 3.1: Weight Stationary

The Figure 3.1 shows the weight stationary. The weight is stored in the processor element, and for different process element, it is loaded with distinct weight data. The input data and partial sum are passed from the local buffer rather than DRAM. Also, the partial sum is conveyed between processor elements. For example, if the we want to calculate the $x1 \times (w0 + w1 + w2 + w3 + w4)$, then the first processor element will do $x \times w0$,and passes the partial sum to next processor element. In the next cycle, the processor will compute $x \times w1$. Also, the second processor will receive the partial sum passed from the first processor element and it plus $x \times w1$ to be a new partial sum. The new partial sum will be passed to the third processor element. Finally, the total sum will be produced in the fifth processor element, which is $x \times w0 + x \times w1 + x \times w2 + x \times w3 + x \times w4$.

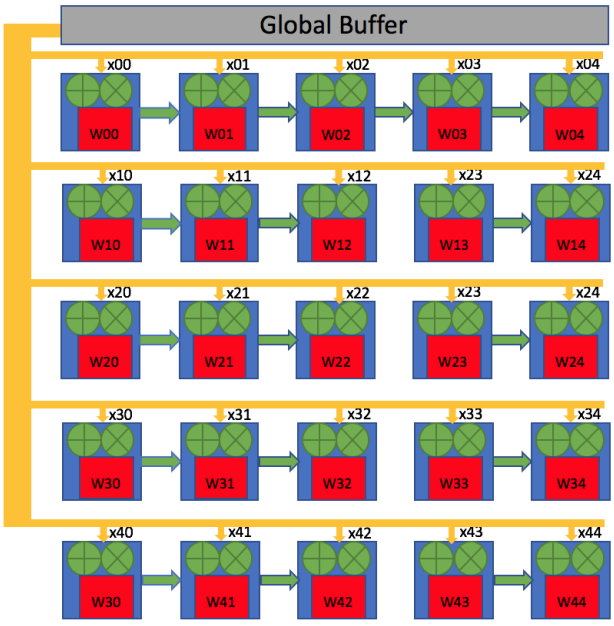

Figure 3.2: 5×5 Convolution In Weight Stationary

Some research works have used this architecture design, such as Neuflow [8] and PRIME [6]. When we use this architecture for deep learning's computation, the matrix computation can be translated to 2D weight stationary. Different rows of weight matrix can be stored in different rows of processor element array. And the input data will be loaded to PE array at different cycles. The partial sums are passed in the horizontal direction, and the final result can be reached at the end of the row.

For a 5×5 matrix convolution, like the Figure 3.2, the weight is stored in processor elements, and the input data x_{ij} will be passed to each row of processor element array. For the convolution computation, the weight matrix will slide through the whole input matrix. So if the weight data is stored in the processor element, there is no need to reread the same weight, which means that the weight data can be reused. Also, instead of getting data from DRAM directly, the weight stationary uses a global buffer to store the data. The cost of accessing memory can be reduced largely.

3.2 Output Stationary

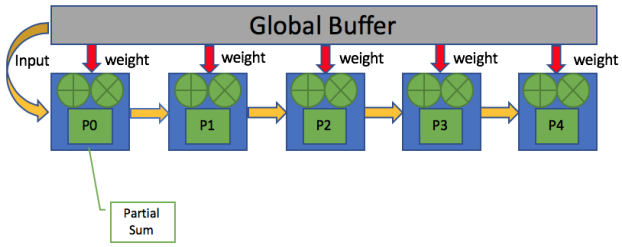

Figure 3.3: Output Stationary

Compared with weight stationary, the output stationary architecture is going to store the partial sums in process elements rather than weight data. And the partial sums will be added up at the end of the computation process. Like the Figure 3.3, if we want to calculate$[x0\,x1\,x2\,x3\,x4] \times [w0\,w1\,w2\,w3\,w4]^T$ by using output stationary strategy. Then w0, w1, w2, w3, and w4 will be passed to the five processor elements in different cycles. Also, these five processor elements can receive x0, x1, x2, x3, and x4 accordingly. After the final computation, $x4 \times w4$ is finished. The partial sums $x0w0, x1w1, x2w2, x3w3, x4w4$ are stored in the five PEs as P0∼P4. The partial sums will not be added up in PEs which is different from weight stationary whose partial sums will be accumulated between PEs.

When we apply this architecture into convolution computation which is the core process of CNN(convolutional neural network), the weight and input can be read from the buffer and then do the computation in processor element. In this paper, we choose this strategy for our systolic array architecture. Also, there is some other research using output stationary such

as ShiDianNao [7] and Peemen [15]. The major advantage of output stationary versus weight stationary is that it can save the energy of reading and writing partial sums data between PEs. Compared with weight data, the partial is normally larger, which will cost more to be passed. So switching from transporting partial sums between PEs to storing partial sums, the energy efficiency and performance can be improved.

3.3 No Local Reuse

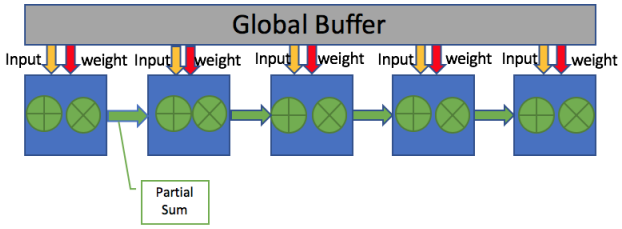

Figure 3.4: No Local Reuse

TPU [11],DaDianNao [5] and DianNao [4] use the no local reuse architecture like Figure 3.4. Compared with output stationary and weight stationary mentioned above, the process element in no local reuse architecture doesn't store the data anymore instead using a large buffer to store the data. In other words, this strategy uses a large global buffer as the shared memory, which can reduce the cost of DRAM access.

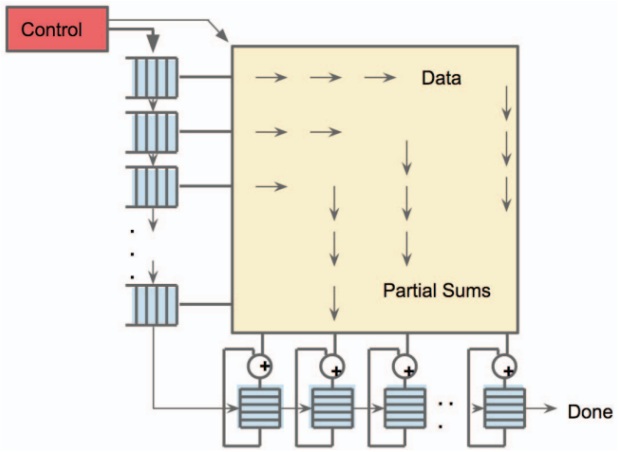

Figure 3.5: TPU Architecture [11]

TPU's architecture like figure3. The input data come from the left side and flow to the right side. The weight data are transported to every PE and will be updated for every

computation round. By using this architecture, the memory space in PE is pretty small, which means that the chip can be made in smaller size even the PE array is vast.

3.4 Row Stationary

The Eyeriss [3] put forward an ASIC which has improved on power efficiency and performance primarily. The core idea is going to reducing the data movement and changing the data source. As we are known, fetching data from DRAM cost much more compared with getting data from SRAM. Also, from their simulation result, the cost of passing data from the Register file should be the least, from Processing Element to Processing Element should be second least. And the process of transporting data from Buffer to PE will cost more than from PE to PE. Under this circumstance, to reduce energy cost and improve performance, we need to minimize data movement, which means that increasing data reuse is required. Also, if data come from high-cost source change to low-cost sources, such as getting data from SRAM instead of DRAM, the total cost will be reduced.

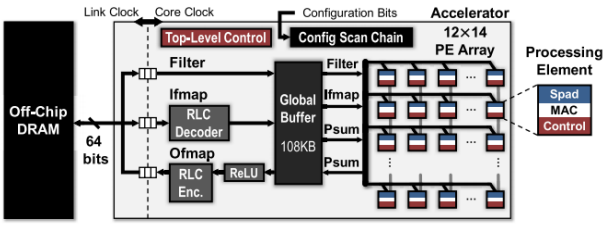

Figure 3.6: Eyeriss System Architecture [3]

Their PE(processing element) has three parts: Spad, MAC and Control. The Spad is a kind of memory unit, which is 0.5∼ 1 kb. It can store results and input data. The MAC unit can do multiplication and accumulation. The Control part is to control the model of Processing Element.

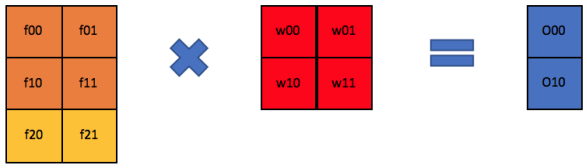

Figure 3.7: Convolution With Stride = 1

The data movement in Eyeriss is using Row Stationary instead of systolic array. In this strategy, the whole row of the kernel matrix will be passed to the same PE row. For example,

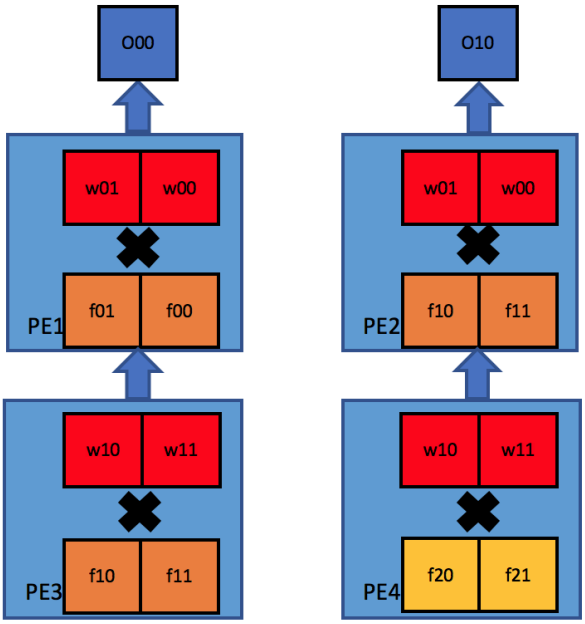

Figure 3.8: Row Stationary

as shown in Figure3.7, the input size is 3 ×2, and the weight size is 2×2. When stride equals to 1, it will generate O00 and O10.

For the row stationary, like Figure 3.8, the PE1, and PE2 get the first row from weight matrix. And the second row of weight matrix is passed to PE3 and PE4. On the other side, the 1st row of the input matrix is only delivered to PE1. And the 2nd row is passed to PE2 and PE3. PE4 gets the last row. All of these data are stored in the Spad unit.

When data are all loaded to PEs, they will do multiplication and accumulation. Every PE will receive its own partial sum, such as (w01×f01 + w00 ×f00) in PE1. And then PEs need to add these partial sums following the direction of the blue arrow to get final result O00 and O10.

Chapter 4

Reconfigurable DNN accelerator

4.1 The Software Design-Krycek

As we have mentioned before, we need a simulator to evaluate configuration candidates. But before we could find out the most suitable configuration for our architecture, we need to generate these configuration lists. So the Krycek will have the feature to create the configuration data. However, if we want to know how to get the data for DNNs, we need the information of DNNs. And then a converter that could take DNNs as input feeds the DNNs to Krycek. After this process, the configuration generator will produce a bunch of configuration lists.

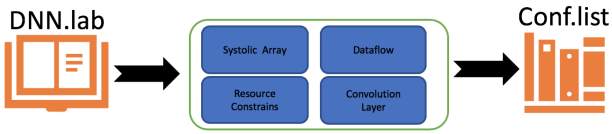

Figure 4.1: Configuration Generator

4.1.1 Generate Configuration Candidates

The configuration generator is like Figure 4.1, the central part of the configuration generator includes systolic array, Dataflow, Resource Constraints, and convolution layer part. Each part will have separate functionalities but also combine to get the final result. The DNN library should have some DNNs such as VGG11, AlexNet, GoogleNet, etc. Also, other deep neural networks could be added into the library in the form of Figure 4.2.

The DNN library contains a lot of popular deep neural networks. For example, as in Figure 4.2, the input, weight and kernel information should all be stored in the library. When the

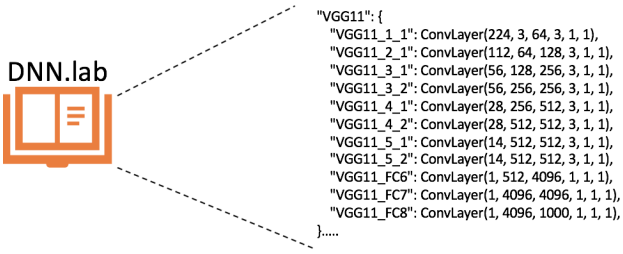

Figure 4.2: DNN Library

DNN is called, this information will be passed to the next process.

After we know DNNs, we need to pass them to Krycek's function. Before we could feed DNNs into our generate function, we need to specify the hardware part. The different board will be equipped with various features. For example, the Figure 4.3 shows Pynq board, which is a kind of Xilinx's FPGA. It has 53200 LUTs(look up table) and 106400 Flip-Flops. But for Figure 4.4, it is also an FPGA from Xilinx company. AC701 board has 134600 LUTs and 269200 Flip-Flops. The hardware board with more resource such as memory and reconfiguration ability will decide how much space we could do the dynamic reconfiguration. So when we receive the DNNs, we still need to add resource information before we can generate configuration candidates. In Krycek, we could decide the number of PEs, the memory, and the FIFO size, which are decided by the chosen board.

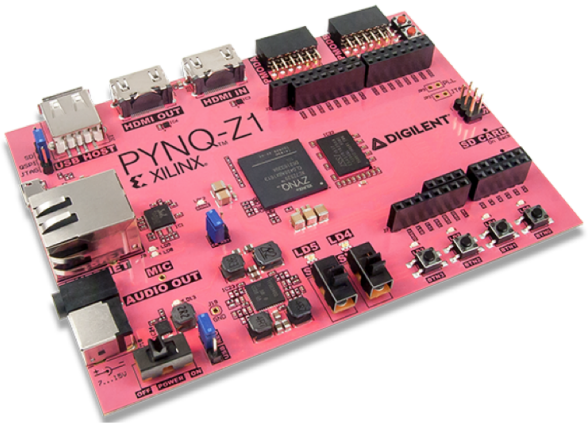

Figure 4.3: Xilinx Pynq Board

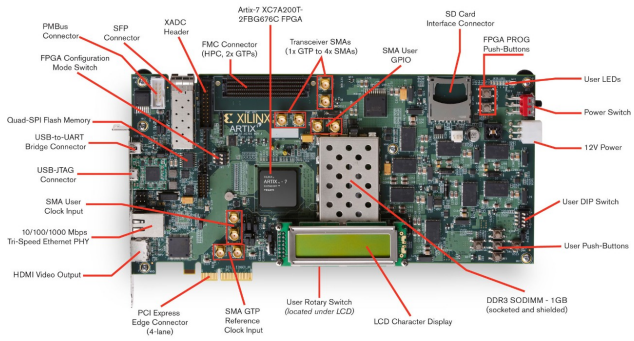

Figure 4.4: Xilinx AC701 Board

In this thesis, we will mention the systolic array for a lot of times because we have to build our architecture on the foundation of systolic array design. The DNN library can store DNN's information. However, we need a series of functions that could do some preparation work for generating configuration data. The systolic array is to clarify the underlying architecture of design. Like in Figure 4.5, the systolic array class in Krycek will collect the hardware

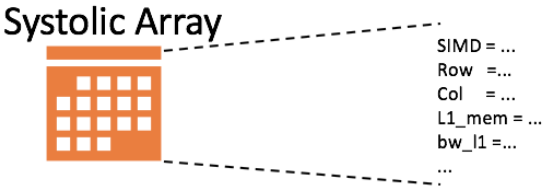

Figure 4.5: Systolic Array Class

information and then sort this information that will be passed to the next function. The values of SIMD, ROW, Column, and FIFO size, etc. represent how the system architecture is.

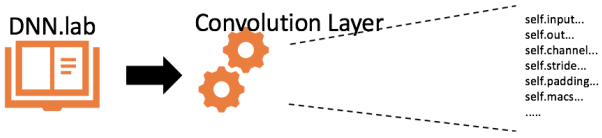

Figure 4.6: Convolution Layer Class

Till now, we already get information about DNN and hardware board. We still need the algorithm of the deep neural network, which will help us to count how many computations will happen in total based on the dnn.lib. And then these results will be stored in convolution layer class, like Figure 4.6.

After previous prepare work has already been done, it's time to get the configuration candidates. We need to sweep all of the possible choices under known constraints. During this process, we are going to change the value of SIMD, Row, and Column, which can decide the shape of our PE array. In the generation process, we are not going to evaluate configurations but produce configuration lists.

Algorithm 1 Generate Configuration

```
1: procedure Loading Information
2: def Dataflow(hardware,layer,dnn,R,L,C):
3:     partial_row_mem = hardware.num_rows * (layer.kernel_size + 1)
4:     partial_col_mem = hardware.num_lanes * layer.kernel_size * layer.kernel_size
5:     batch_size,row_mem, col_mem, temp_mem = Systolic(partial_row_mem, partial_col_mem, dnn)
6:     fifo_size = row_mem + col_mem + temp_mem
7:     if fifo_size > hardware.fifo then
8:         batch_size = 1, fifo_size = row_mem + col_mem
9:         if fifo_size >hardware.fifo then
10:            batchsize = FindMax(batchsize)
11:    tmp_in,tem_out,tmp_par = FindAllPossibleCombinationsLessThanL1_mem
12:    times = FindTimes
13:    iterations = FindIterations
14:    return row_mem, col_mem, times,cycls,R,L,C....
15: procedure Generate data
16: def generate(args)
17:    dnn = args.dnn
18:    R,L,C = Systolic(dnn)
19:    hardware = Constraints(arg.board)
20:    Configuration_list = Combinations(R, L, C, hardware.l1.hardware.fifo)
21:    Configuration_candidates = Dataflow(Configuration_list)
```

Like in algorithm1, the Dataflow function takes all prepared data and then return the details of configuration. And in the generation function, when it will get the configuration list then pass the file to Dataflow function to produce configuration candidates.

4.1.2 Evaluation

The previous section introduces the process of generating configuration candidates. However, we need to filter unqualified configurations when we apply metrics constraints. Also, the generation process will give all possible configuration candidates. We need to think about a more realistic factor when we chose the best configuration for our neural network.

When we find the better configuration for DNN layer versus static configuration, we still need to think about the reconfiguration cost. Because in hardware running process, any action

will cause delay and energy cost. Before we could decide whether to take reconfiguration, we need to add the cost of reconfiguration and compare it with static's configuration.

Algorithm 2 Evaluation

Loading Configuration Candidates
Input:$DNN, metrics\_constraints, Reconfiguration\_cost, Configuration\_candidates$
Basic_configuration = $FindConfigurationFitForAllLayers(Configuration\_candidates)$
for $configuration\_candidates[i].metrics > metrics\_candidates$ **do**
    configuration_candidates[i].delete
▷ Get the new configuration Candidates
reconfiguration_list= $Sorted(configuration_candidates)de$
for everylayer **do**
    if $configuration\_candidates.cost + reconfiguration\_cost < previous\_configuration.cost$ **then**
        $DoReconfiguration$
    else
        $KeepSame$
Return $Configuration\_map$

In algorithm2, it shows the process of finding the best configuration list for DNNs and whether taking reconfiguration. The first step is to delete the configuration candidates that require higher metrics such as power and performance than given parameters constraints. Then we find out a base configuration that will fit for all DNN layers. And we begin evaluation at the base configuration for every DNN layer. If this layer's total cost after reconfiguration plus reconfiguration cost is less than the cost under the previous configuration, we will choose to take the reconfiguration. So the control signals passed to the hardware part are changed to meet new configuration. However, if we find that cost under reconfiguration plus reconfiguration cost is higher than cost under the previous configuration, we will not change the configuration and keep it as same as the last configuration. It means that even the cost under reconfiguration will lower than under the previous configuration, we need to count cost overhead. So the hardware part will not change if not taking reconfiguration.

4.1.3 Results

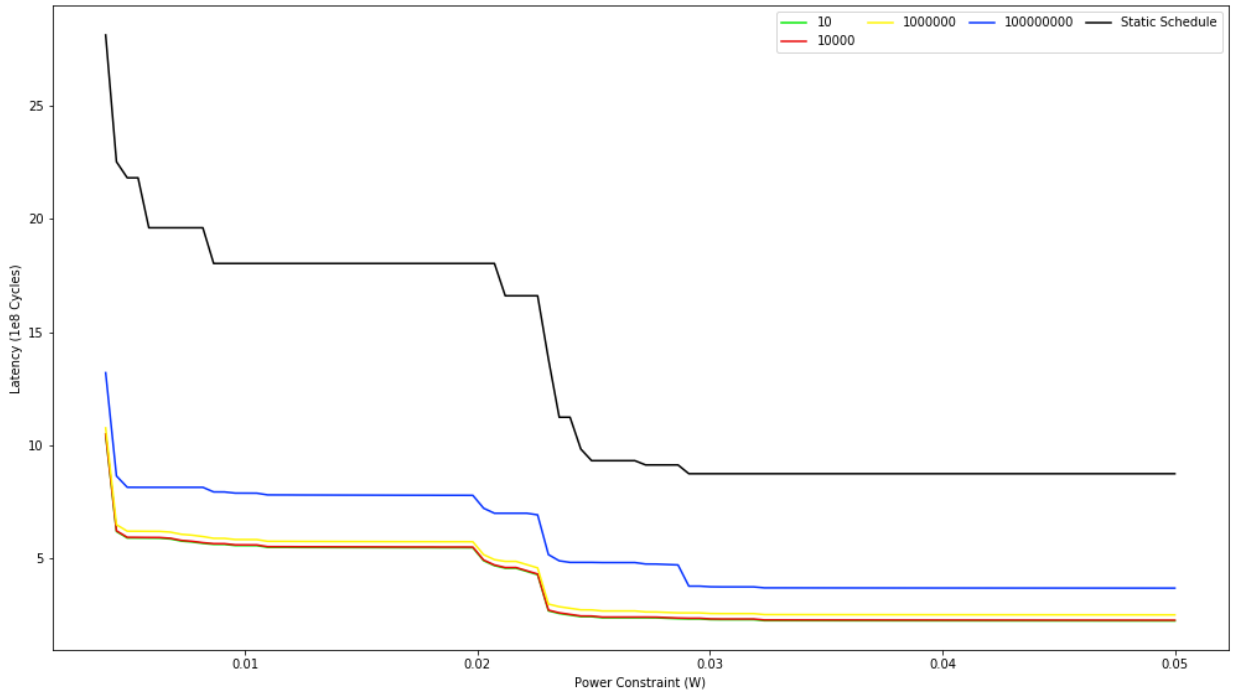

Figure 4.7: Static VS Reconfiguratable

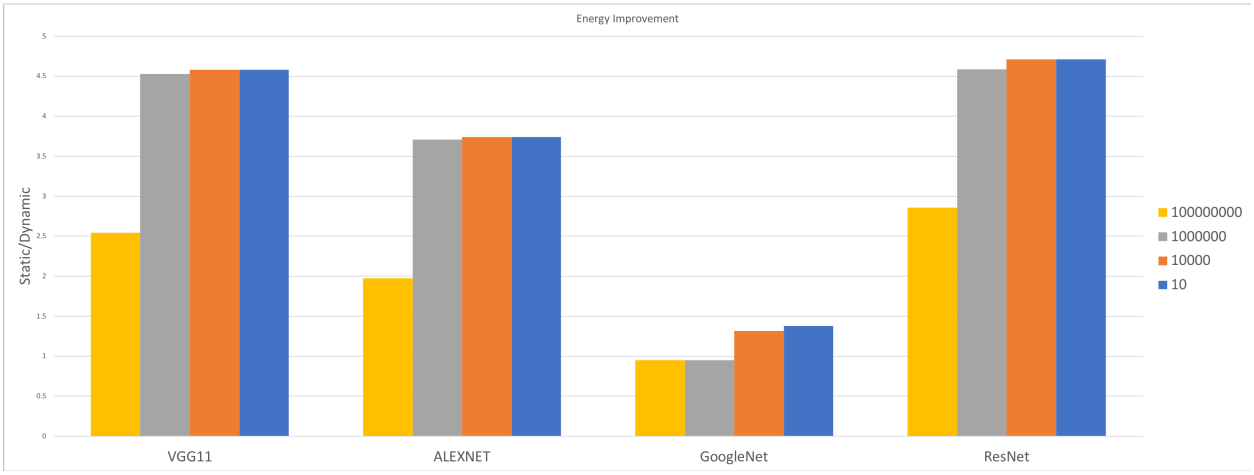

Figure 4.8: Energy Improvement

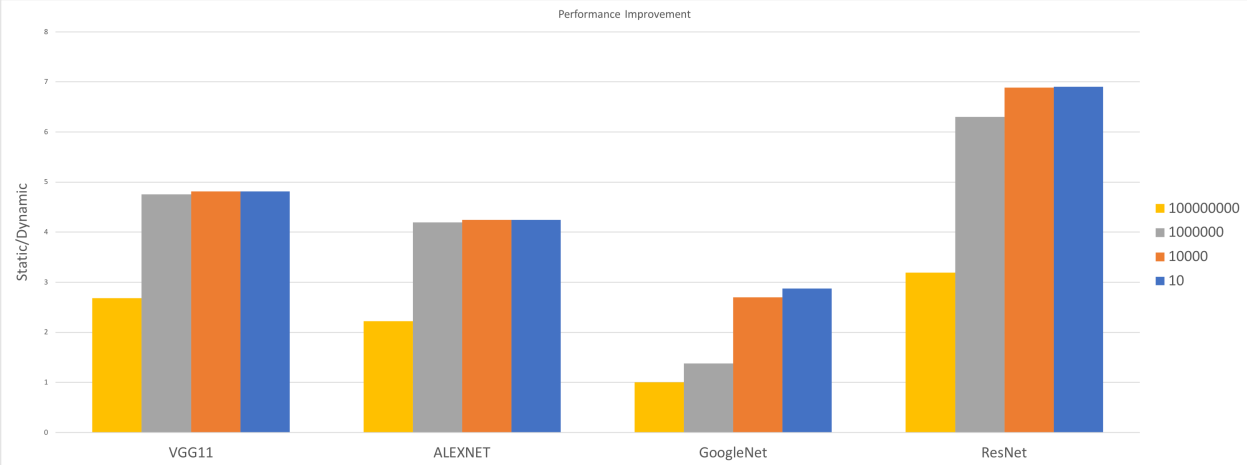

Figure 4.9: Performance Improvement

The Figure 4.7 shows when we change the power constraints for our network, the tendency of performance. As we increase the power constraint, it means that our system could have more chance to find out the better configuration in terms of performance. Because the total number of configuration candidates is fixed, when the power constraint is large, more configurations will be kept after filtration. In the hardware part, the high power constraint also makes the system have better performance at the price of costing more energy.

The different color lines expect the black one represents various hardware boards. The black line shows the cost of using a static configuration. The costs to do reconfiguration in distinct boards are different. For examples, when the reconfigurable DNN models are running on an ASIC or particular purpose hardware part, they will take fewer cycles to do reconfiguration, which means cost less. However, if we run our models on a general purpose board, more cycles are needed for reconfiguration during DNNs executions. From the Figure 4.7, even the cycles that are required to do reconfiguration are vast, the dynamic reconfiguration still has an advantage versus static configuration at different power constraints.

The Figure 4.8 shows the energy improvement of various boards versus static configuration under different DNNs when we apply the dynamic configuration. From this picture, we see the improvements in ResNet50 is the largest. And the GoogleNet has the smallest improvement. Also, when we change the cycles needed for reconfiguration, the improvement change accordingly. The reconfiguration that needs the most cycles will bring the least energy improvement. Even when the cycle is too large, there is no improvement for DNNs such as GoogleNet. It means that if the hardware board is not powerful enough, it will need more cycles for reconfiguration, which leaves less chance for energy improvement. For the Figure 4.9, it shows the performance improvement when the dynamic configuration is applied compared with static configuration. The tendency of performance improvement is similar to energy improvement. Different DNNs will have definite improvement, and higher reconfiguration cycles will cause lower performance improvement.

4.2 The Hardware Design-CNN-SAC

The Krycek will give a bunch of configuration candidates. After we get the constraints in metrics that we care about such as energy efficiency, hardware resource, and performance, we can get the configuration choice from the generated configuration candidates list. As we have known, there is no free lunch, and there will be a lot of trade-off between the choice of the metrics. The Krycek is a tool that can help us find the best option under the constraints. However, the software is just simulation and estimation, and we need to get a tool to verify it. Under the request of this desire, mapping DNNs architecture to hardware such as FPGA and then running testbenches on it is needed. So we need to build this system architecture first by using Chisel. And this architecture design should have the feature that can dynamically reconfigure during DNN execution.

4.2.1 PE Design

For dynamic reconfiguration, we use the Multiplexers to control which configuration is chosen. In the Chisel design, the whole PE array is made of a bunch of basic modules. And every basic module is like a black box that can receive inputs and give outputs. For the primary PE in Chisel, it covers three parts: Control, Memory and Computational part, like Figure 4.10.

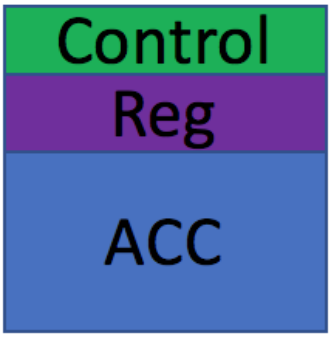

Figure 4.10: Basic PE

The control part is going to switch the implemented configurations during the DNN layers execution. Through changing the source of data, we can change the shape of the PE array, which will be more suitable for different layer's input dimension. The register file in PE is designed for storing partial results. Compared with getting data from DRAM or SRAM, passing data through register will be more energy efficient and has higher performance. Also,

in hardware design, the input data reaching to the PE should be stored before this data can be passed to next PE. The computational unit can do multiplication and accumulation during execution.

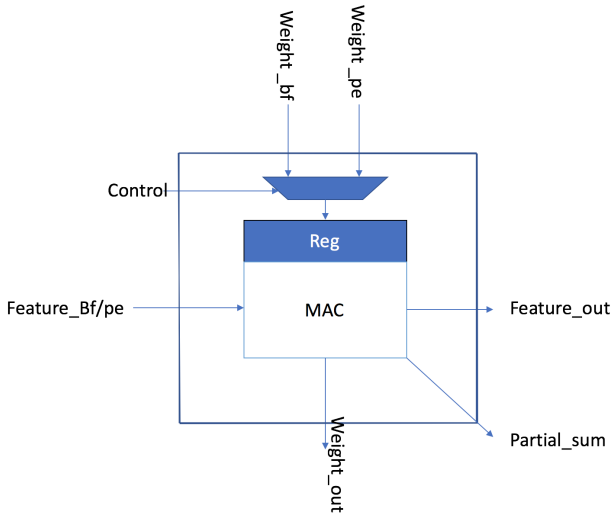

Figure 4.11: Inside Of PE

The inside of PE is like Figure 4.11, using the control signal to control the weight data coming from neighboring PE or Buffer directly. However, for the input feature data, it always comes from its left side, which should be buffer or PE. Also, the weight and feature should be used by next PE. For feature data, it will be passed to its right side PE at next cycle. And weight data will flow to bottom PE at next cycle. The result generated by the multiplication between feature and weight will be added and stored in the register file. The diagonal accumulation happens after all PE finishes its tasks, and the final result for convolution can be obtained.

4.2.2 SIMD

The SIMD(single instruction multiple data) [14] is a kind of parallel computing strategy. By using SIMD in computing, the performance will be largely improved. In this thesis, we also use this trick to enhance our computing performance.

If there is four PEs, like Figure 4.12,Figure 4.13 and Figure 4.14. When SIMD = 1, in Figure 4.12, the PE1 receives the feature data first. After PE1 finishes its computation, it will pass data to PE2. However, in Figure 4.13, PE1 and PE2 get features at the same time but receive different weight data. After they finish using the feature data, PE3 and PE4 will receive feature data. As shown in Figure 4.14, SIMD=4, it means that four PEs receive the same feature data at the same time. This strategy will be pretty useful in DNN computational process. Because the dimension of data in different layers is dramatically different. Through

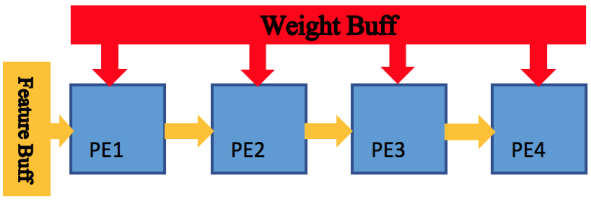

Figure 4.12: SIMD = 1

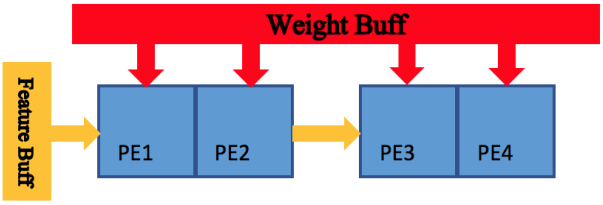

Figure 4.13: SIMD = 2

changing the value of SIMD, we could achieve higher throughput and improve our design's performance. When the data has large height and width, but small depth, the design with small SIMD will be a better choice.

The SIMD strategy allows us to do reconfiguration. When the hardware is chosen, the number of processor elements will be fixed.

$$Row \times SIMD \times Col = The\ number\ of\ PEs$$

Like the above formula, we could shift the value of Row or SIMD to change the shape of the PE array. However, the value of Col could be fixed when DNN is chosen, because the value of the column equals the value of kernel's row. For example, when the kernel size is 3× 3, the col should be 3. But when the kernel size is 11× 11, the value of the column should also be changed to 11.

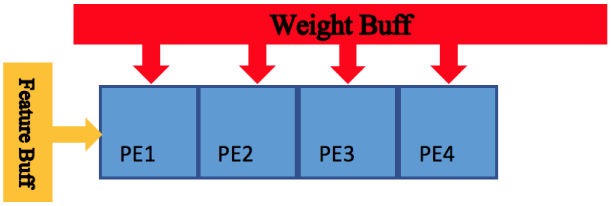

Figure 4.14: SIMD = 4

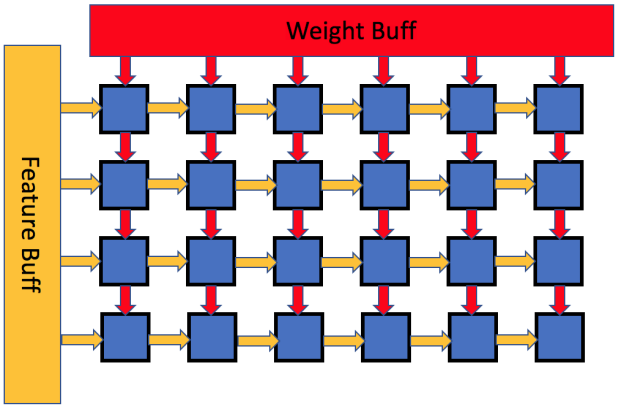

Figure 4.15: Base Architecture

4.2.3 PE Array Design

The Figure 4.15 is an example that there is 24 PEs. According to the previous formula, when we choose the weight size is 3×3 and change value of Row and SIMD, there will be several combinations. Also, in realistic design, we need to think of the complexity of wire connection. The more combinations mean the architecture is more complex for more wire connections. So as the number of PEs goes up, it will be impossible to implement all combinations of (R, L, C). Under this circumstance, we prefer to choose the value of R and L that is the power of 2. Also, we want to use PEs as much as possible. It means that we are not going to make some PEs idle if we can. For the Figure 4.15, it has 24 PEs. The combinations that will be mapped to the hardware part is just four like table4.1

Row	1	2	4	8
SIMD	8	4	2	1

Table 4.1: ROW and SIMD

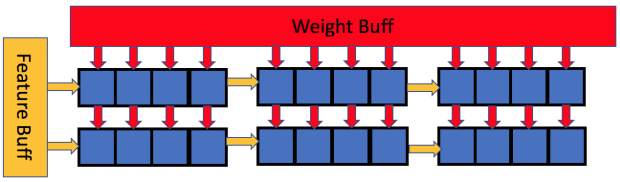

Figure 4.16: SIMD = 4

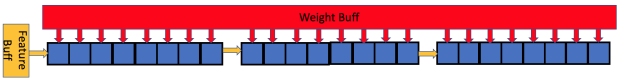

Figure 4.17: SIMD = 8

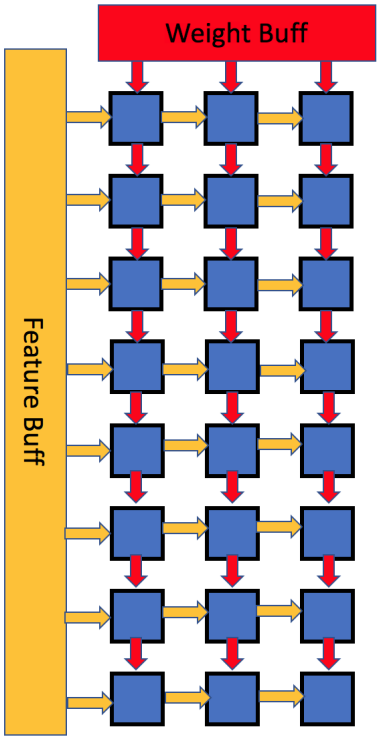

Figure 4.18: SIMD = 1

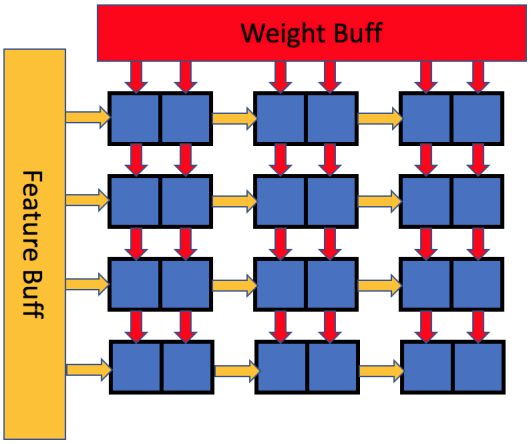

Figure 4.19: SIMD = 2

The Figure 4.18,Figure 4.19, Figure 4.16 and Figure 4.17 show the different configurations of this architecture design. The column is fixed at three because weight size is determined. From these four figures, we can see the shape of the PE array changing largely. Also, for PE in the same SIMD and the same row, the data arrive at PEs at the same time. But the weight data arriving at different PEs are distinct while the feature data arriving are the

same. Also, for the total number 24, there will be other combinations. The reason why filtering these combinations is that we want to keep as many as possible PEs busy. So the combinations like $(Row, SIMD, C) = (1, 2, 3)$, which a part of PEs will be idle, are not kept. The other reason for choosing the value of Row and SIMD that are pow of 2 can make the reconfiguration easier compared with choosing arbitrary numbers.

4.3 Design In Chisel

In the previous chapter, we show how to build a multiplier in Chisel, which requires us specifying the input, output, connection, bitwidth, etc. So when we use Chisel to design our module, we need to think of designing hardware rather than coding in software. It will be better if we could build PE first and then call this module as many times as we want instead of making the whole system at one time.

4.3.1 Basic In Chisel

For the basic PE module, as we show in Figure 4.11, we need a Mux to control the weight source. Through this strategy, the PE array's shape can be changed. Also, in the hardware design, we need to store the input data into the register file so that the input data can be accessed by next PE. Without this action, all the PE will be connected in wire, which will not be the systolic array.

PE.Scala

```
1  import chisel3._
2  import chisel3.util._
3  class PE(inputsize:Int,bitwidth:Int,bitwidth2:Int) extends Module{
4    val io = IO(new Bundle{
5      val control = Input(Bool())
6      val inputpixel = Input(Vec(inputsize,UInt(bitwidth.W)))
7      val inputweight=Input(Vec(inputsize,UInt(bitwidth.W))) // get the weight
           ↪ from the pe
8      val inputweightBF=Input(Vec(inputsize,UInt(bitwidth.W)))
9      val out = Output(Vec(inputsize,UInt(bitwidth2.W)))
10     val total = Output(UInt(bitwidth2.W))
11     val pixelout = Output(Vec(inputsize,UInt(bitwidth.W)))
12     val weightout = Output(Vec(inputsize,UInt(bitwidth.W)))
13   })
14   val myreg0=RegInit(VecInit(Seq.fill(inputsize){0.U(bitwidth.W)}))
15   val myreg1=RegInit(VecInit(Seq.fill(inputsize){0.U(bitwidth.W)}))
16   val myreg2=RegInit(VecInit(Seq.fill(inputsize){0.U(bitwidth2.W)}))
17   val myregtotal=RegInit(0.U)
18
19   for (i<-0 to inputsize-1){
20     myreg0(i) := io.inputpixel(i)
21     myreg1(i) := io.inputweight(i)
22     myreg2(i) := io.inputweightBF(i)
23   }
```

```
  when(io.control) {
    for (i <- 0 to inputsize - 1) {
        io.out(i) := myreg0(i) * myreg1(i)
        io.weightout(i) := myreg1(i)
        io.pixelout(i) := myreg0(i)

    }
    io.total := io.out.reduce(_+_)
  }
    .otherwise {
      for (i <- 0 to inputsize - 1) {
          io.out(i) := myreg0(i) * myreg2(i)
          io.pixelout(i) := myreg1(i)
          io.weightout(i) := myreg2(i)
      }
      io.total := io.out.reduce(_+_)
    }
}
```

In the code PE.Scala, it shows the details of designing the primary PE. The design is based on the Scala language. The *'io'* is the interface of the PE module. And we need to declare the input size and output size. In the IO block, we should describe input and output by using *Input* and *Output*.

For the algorithm, this is just a basic module, and its task is taking the input then storing the output. Its behavior is controlled by the *'control'* signal. When the control signal is true, the PE takes weight data from the previous PE module. Or when the control signal is false, the weight data arriving at PE comes from weight Buffer.

Chisel provides a feature for us to test the functionality of our design, like PETester.Scala. We need to give the input data and expected output data. Then Chisel could check whether the expected output is the same as the generated output data.

PETester.Scala

```
class PETester(c:PE_VGG) extends PeekPokeTester(c){
    val input_pixel = Array(1,2,3,4)
    val input_weight = Array(2,3,4,5)
    val expected_out = Array(2,6,12,20)
    val expected_total = 40

    poke(c.io.control, true.B)
    for(i<-0 to 3){
        poke(c.io.inputpixel(i),input_pixel(i))
```

```
10          poke(c.io.inputweight(i),input_weight(i))
11      }
12      step(1)
13      for(i<-0 to 3){
14          expect(c.io.out(i),expected_out(i))
15      }
16      expect(c.io.total,expected_total)
17
18  }
19  val tester = Driver(()=> new PE_VGG(4,8,16)){
20      c=>new PETester(c)
21  }
22  assert(tester)
23  println("success")
```

Also, we could get the waveform from Verilator related to our architecture design. Through checking the correctness of waveform, we can analyze the value and time of input and output to check the correctness of our design.

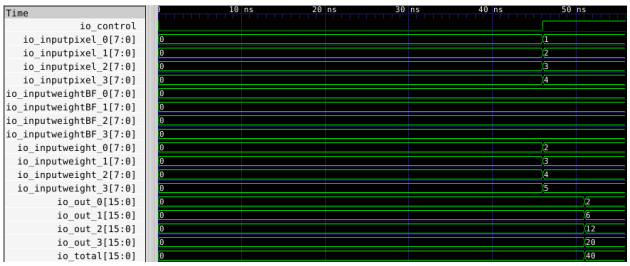

Figure 4.20: PE Waveform

In Figure 4.20, it shows the input pixel, weight, control signal and output. From the waveform, we know that there is one cycles delay between the input and output, which matches with the tester file. Also, in hardware part, it actually needs time to get input data before generating output.

4.3.2 Systolic Array In Chisel

In Figure 4.20, it shows the input pixel, weight, control signal, and output. From the waveform, we know that there is one cycles delay between the input and output, which matches with the tester file. Also, in the hardware part, it needs time to get input data before generating output.

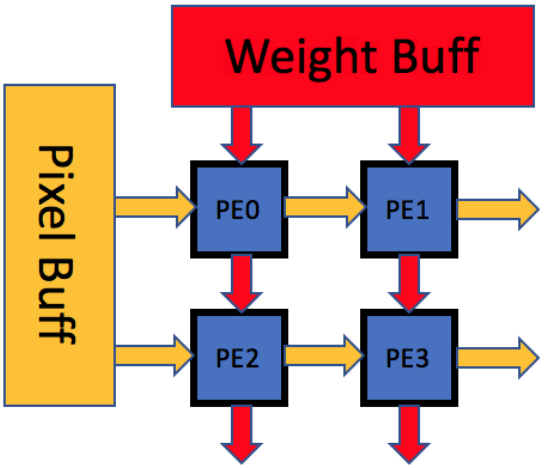

Figure 4.21: Systolic Array Tester

SystolicArray.Scala

```
1  class SystolicArray(inputsize:Int,bitwidth:Int,bitwidth2:Int) extends Module
     ↪ {
2    val io = IO(new Bundle{
3      val inputpixelBF = Input(Vec(2,Vec(inputsize,UInt(bitwidth.W))))
4      val inputweightBF = Input(Vec(2,Vec(inputsize,UInt(bitwidth.W))))
5      val out = Output(Vec(4,UInt(bitwidth2.W)))
6    })
7    val PE_base = Array.fill(4)(Module(new PE_VGG(inputsize,bitwidth,bitwidth2
       ↪ )).io)
8    for(i<-0 to 3){
9      PE_base(i).control := true.B
10   }
11   for(i<-0 to inputsize-1){
12     PE_base(0).inputweight(i) := ShiftRegister(io.inputweightBF(0)(i),1)
13     PE_base(1).inputweight(i):= ShiftRegister(io.inputweightBF(1)(i),1)
14
15     PE_base(2).inputweight(i) := ShiftRegister(PE_base(0).weightout(i),1)
16     PE_base(3).inputweight(i) := ShiftRegister(PE_base(1).weightout(i),1)
17
18
19     PE_base(0).inputpixel(i) := ShiftRegister(io.inputpixelBF(0)(i),1)
20     PE_base(2).inputpixel(i) := ShiftRegister(io.inputpixelBF(1)(i),1)
21
22     PE_base(1).inputpixel(i) := ShiftRegister(PE_base(0).pixelout(i),1)
23     PE_base(3).inputpixel(i) := ShiftRegister(PE_base(2).pixelout(i),1)
```

```
24    }
25    for(i<-0 to 3){
26      io.out(i) := PE_base(i).total
27    }
28
29    for(i<-0 to 3){
30      for(j<-0 to 3){
31        PE_base(i).inputweightBF(j) := 0.U
32      }
33    }
```

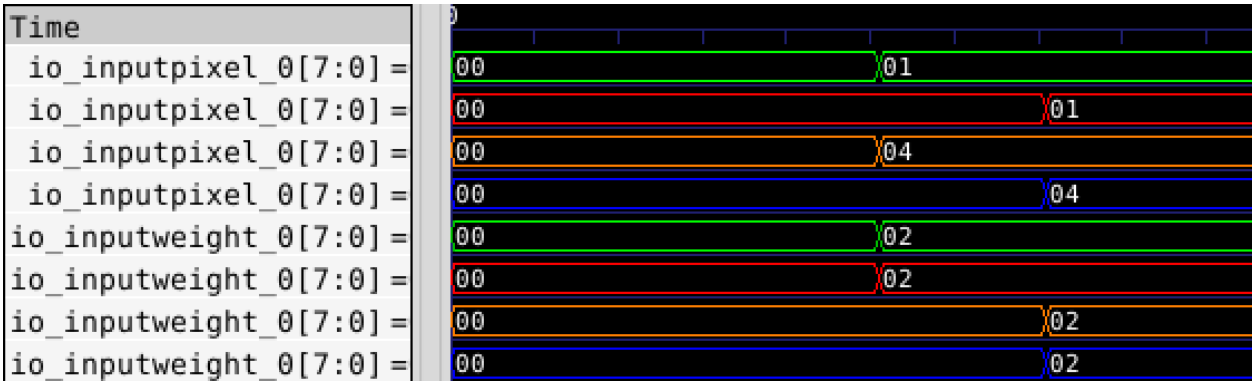

Figure 4.22: Systolic Array Waveform

To test the functionality of systolic array built by Chisel, we use a straightforward 2D systolic array like Figure 4.21. For the systolic array, the PEs receive data at different time. For example, PE2 will receive the same weight data as PE0 but at the next cycle. Also, PE3 will receive the corresponding pixel data as PE2 with one cycle delay. In the code Systolic Array Class, we call the base PE module in line 7 and then connect them in line 11 $\sim$ 24. In the base PE, we use the control signal to control the source of weight data to change the shape of the PE array. For the tester in Figure 4.5, we need to choose the PE2 and PE3 to receive weight data from PE0 and PE1 separately.

The figured4.22 shows the time of 1st data arriving in different PEs. The green, red, orange and blue represent the data in PE0 $\sim$ PE3. There is a one cycle delay between PE2 and PE3, PE0 and PE1 for input pixel. Also for input weight, there is one cycle between PE0 and PE2, PE1 and PE3. These circumstances meet with the functionality of the systolic array.

So from this section, we verify that the systolic array made by Chisel has the correct behavior.

4.4 Dynamic Reconfigurable Systolic Array

The previous sections describe how to use Chisel to achieve our hardware design and test our design's functionality. The process of mapping a DNN to a hardware board is evaluating combinations in Krycek and make it Chisel. So we need to think about the actual board about how many resources we could get and then choose those possible configurations for our hardware design.

The Krycek could provide the initial evaluation of potential configuration candidates. The CNN-SAC can generate the RTL files of our design, which can be synthesized and implemented to hardware boards. Also, we could adjust the parameters of Krycek by using the results of synthesis.

4.4.1 Platform And Benthmark

In this paper, we use the VGG11 as our benchmark to test the whole design. Although ResNet shows the largest improvement in Krycek's results, we don't have the FPGA board that can test the reconfigurability of ResNet for ResNet have larger network size and diversity in the data dimension.

DNN layer	Nin	Nout	Nij	Kij
CONV1	3	64	224 × 224	3 ×3
CONV2	64	128	112 × 112	3 ×3
CONV3_1	128	256	56 × 56	3 ×3
CONV3_2	256	256	56 × 56	3 ×3
CONV4_1	256	512	28 × 28	3 ×3
CONV4_2	512	512	28 × 28	3 ×3
CONV5_1	512	512	14 × 14	3 ×3
CONV5_2	512	512	14 × 14	3 ×3

Table 4.2: Convolution Layers of VGG11

In table2.1, the dimension of input and output change dramatically. It means that if we apply static configuration for every convolutional layer, there should be a conflict between hardware resource and request of software data. However, if we could change the shape of the hardware part when we execute different layers that have colossal data dimension difference, the conflict can be buffered.

The previous sections give an introduction to building a systolic array by calling the basic PE module and connecting those modules in the desired way. We need first to decide the maximum PEs that can be used and then sweep all possible configurations to get configuration candidates.

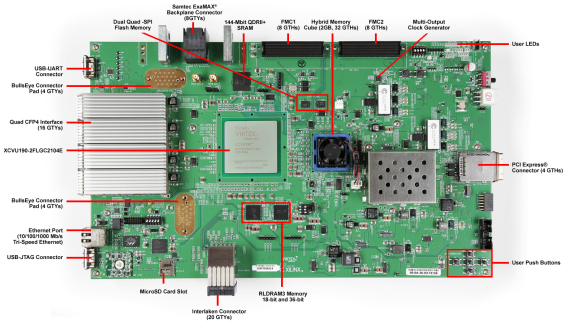

Figure 4.23: Xilinx VCU110 Board

In this thesis, we chose the Xilinx VCU110 as our platform. It has 1074240 LUT elements and 2148480 FlipFlops. We first need to find out the maximum number of PEs that could be mapped to this board. Also, we notice that the kernel size is always 3. So if we choose the number of PEs is times of three, the dynamic reconfiguration will be easier because the value of PE's column equals kernel's row size. When the total number of PEs can be divided by three, the Row and SIMD can be integer according $Col \times SIMD \times Row = PEs$.

PEs =768	SIMD	256	128	64	32	16	8	4	2	1
	ROW	1	2	4	8	16	32	64	128	256
PEs =384	SIMD	128	64	32	16	8	4	2	1	
	ROW	1	2	4	8	16	32	64	128	
PEs =192	SIMD	64	32	16	8	4	2	1		
	ROW	1	2	4	8	16	32	64		
PEs =96	SIMD	32	16	8	4	2	1			
	ROW	1	2	4	8	16	32			

Table 4.3: Configurations Of Different PEs

The maximum number of PEs can be fit in VCU110 is near 800. We choose the 768 as the maximum. Because the column's value is fixed at three, and the values of row, SIMD are pow of two. After the max PE array is chosen, we could also test the PE array that has PEs less than 768. Like in table4.3, we want another three PE arrays to be mapped to the VCU110 board. For these four PE arrays, because the column is kept at 3, changing the value of row or SIMD can get all combinations.

4.4.2 Codes

CNN-SAC.Scala

```
class PE_SystolicArray(inputsize:Int,bitwidth:Int,bitwidth2:Int,pesize:Int) extends Module{
  val io = IO(new Bundle{
  val inputweightBF = Input(Vec(pesize*3,Vec(inputsize,UInt(bitwidth.W))))
  val inputpixel = Input(Vec(pesize, Vec(inputsize,UInt(bitwidth.W))))
  val control = Input(Vec(3,Vec(inputsize,Boolean())))
  val total = Output(Vec(pesize, Vec(inputsize,UInt(bitwidth2.W))))

})
  val PE_VGG_base = Array.fill(pesize,3){Module(new PE_VGG(inputsize, bitwidth,bitwidth2)).io}
  for(i <- 0 to pesize-1){
  for(j <- 0 to 2){
  PE_VGG_base(i)(j).control := io.control(j)(i)
}
}
  PE_VGG_base(0)(0).inputpixel := ShiftRegister(io.inputpixel(0),1)
  PE_VGG_base(0)(0).inputweight := ShiftRegister(io.inputweightBF(0),1)
  PE_VGG_base(0)(1).inputpixel := ShiftRegister(PE_VGG_base(0)(0).pixelout ,1)
  PE_VGG_base(0)(2).inputpixel := ShiftRegister(PE_VGG_base(0)(1).pixelout ,1)

  PE_VGG_base(0)(1).inputweight := ShiftRegister(io.inputweightBF(1),1)
  PE_VGG_base(0)(2).inputweight := ShiftRegister(io.inputweightBF(2),2)

  for (i<- 0 to pesize-1){
  for(j<- 0 to 2){
  PE_VGG_base(i)(j).inputweightBF:= ShiftRegister(io.inputweightBF(3*i+j),j)
}
}
  for (i<- 1 to pesize-1){
  PE_VGG_base(i)(0).inputpixel := ShiftRegister(io.inputpixel(i),i)
        }
}

  for (i<- 1 to pesize-1){
  for(j<- 1 to 2){
  PE_VGG_base(i)(j).inputpixel:= ShiftRegister(PE_VGG_base(i)(j-1).pixelout ,1)
  }
}
```

```
38 }
39
40   for (i<- 1 to pesize-1){
41   for(j<- 0 to 2){
42   PE_VGG_base(i)(j).inputweight :=ShiftRegister(PE_VGG_base(i-1)(j).
         ↪ weightout,1)
43 }}
44   //connect the output
45   for(i <- 0 to pesize-1){
46   for(j <- 0 to 2){
47   io.total(i)(j) := PE_VGG_base(i)(j).total
48 }
49 }
50   for(i<-1 to pesize-1){
51   for(j<-1 to 2){
52   when(PE_VGG_base(i)(j).control){
53   io.total(i)(j) := PE_VGG_base(i)(j).total + io.total(i-1)(j-1)
54 }.otherwise {
55 }
56 }
57 }
58 }
```

The previous chapters describe the steps of building the basic PE module, which requires specifying the hardware details like bitwidth, input size, and output size. And then the section of the systolic array in Chisel introduces that the systolic array built by Chisel has the correct functionality. The codes above are the details of building a scalable Systolic Array by calling the basic PE modules and then connecting them in the desired way to achieve the final CNN-SAC design.

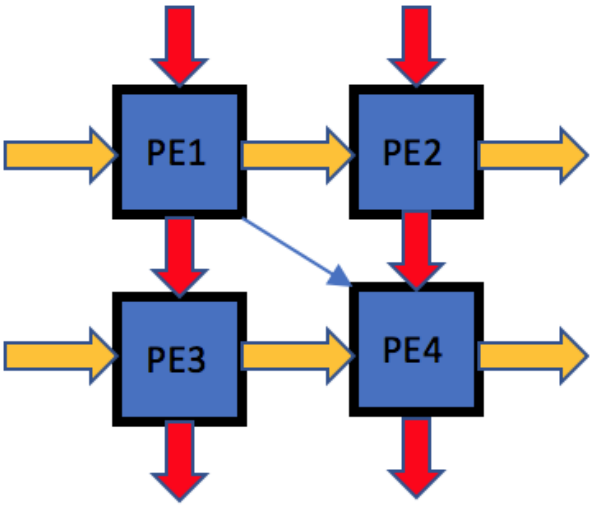

Figure 4.24: Diagonally Sum Up

The previous chapters tell how to change the shape of PEs array. It's going to use the Mux to control weight data source of the processor element. However, there is still one thing left for us to think carefully, the diagonal accumulation. After all computation processes have been finished, the partial sums are stored in processor elements. And then we need to add all of them together diagonally to get the final results. However, when the PE array's shape is changed, it means that some diagonal connection will disappear. Like Figure 4.24, the partial sums in PE1 and PE4 will be added up at the end of one computation round. However, when the shape is changed like that PE3 receives weight data from buffer directly, so the partial sums of PE1 and PE3 shouldn't be summed up, and the diagonal connection between them will not be useful. To solve this issue, we can also use the mode of Mux to control the partial sums in PEs to be added diagonally or not. Like shown in code Systolic Array Class, line45 $\sim$ 54, when the previous PE is connected with PE above it, the diagonally add up will happen, or the partial sums of diagonal PEs will not be summed up.

4.5 Results

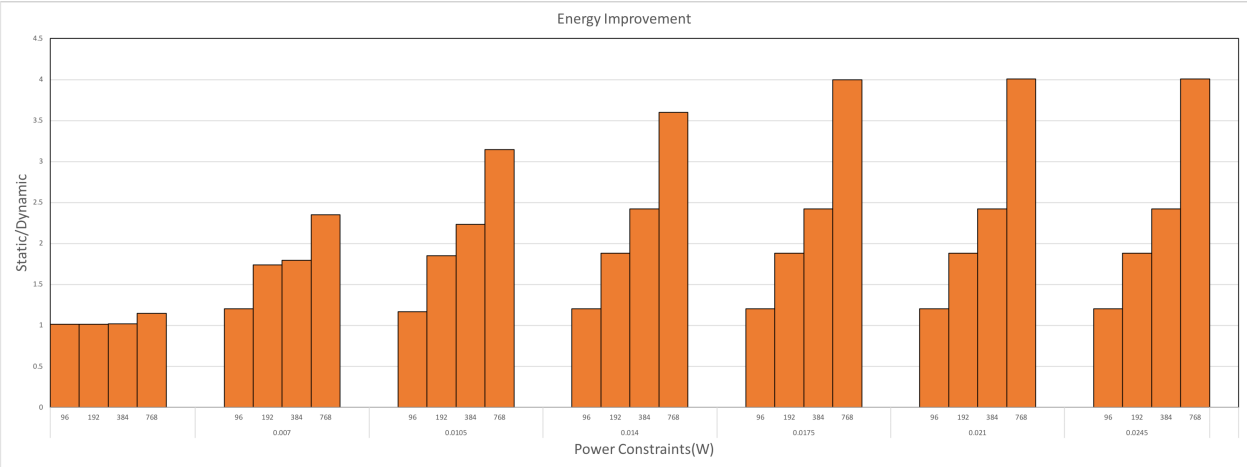

Figure 4.25: Reconfiguration Energy Improvement

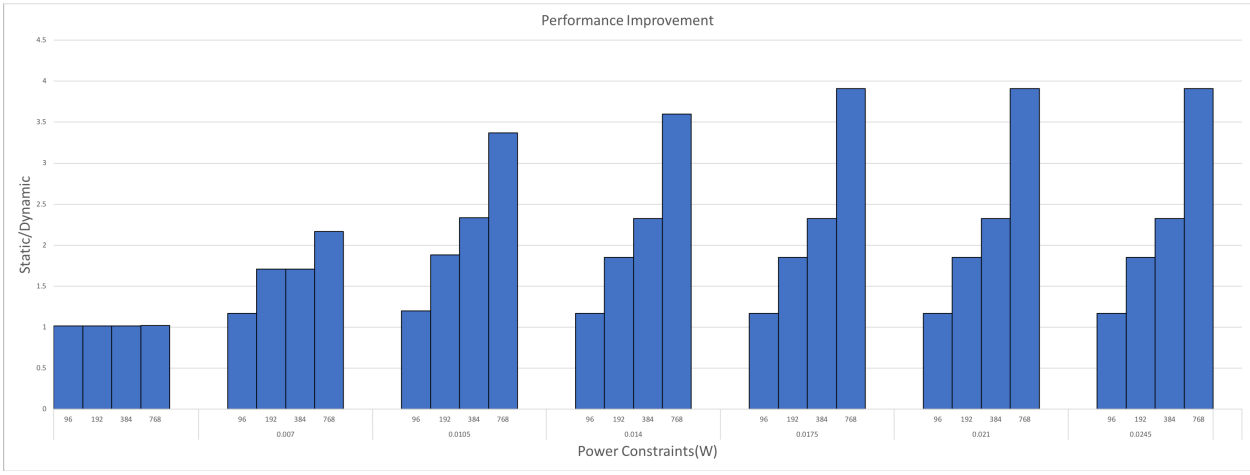

Figure 4.26: Reconfiguration Performance Improvement

The previous sections give the information for reconfiguration during DNN execution. And we choose the VCU110 as our test platform. The table4.3 shows the different PE array that we are going to run on board. The VCU110 board can cover about 800 processor elements. Under this constraint, we choose 768 as the largest PE array to be mapped on the board.

Through changing the SIMD, Row and Col, we can change the shape of PE array to meet the different data dimension's requirement of DNN layers. And the VGG11 is the network we use for a test. Its data shows in table2.1. With all of these prerequisites, then when we apply different power constraints for our network, the energy efficiency and performance

improvement of dynamic reconfiguration compared with static configuration is obtained like Figure 4.25 and Figure 4.26.

From Figure 4.25 and 4.26, the horizontal axis shows the size of PE and different power constraints. The vertical axis is the improvement calculated by (static configuration's energy / dynamic configuration's energy efficiency) and (static configuration's performance / dynamic configuration's performance). At the group one, when the power constraint is pretty small about 0.0035 Watts per layer, the energy and performance improvement are tiny. The reason for this circumstance is that the configuration choice is mainly controlled by the power metric, which means the PE array has less freedom to choose the better configuration. So for the first group, even the size of the PE array is mainly changed, the improvements between different PE array are nearly the same.

As the power constraint increases, the improvement of energy and performance go up. The configuration candidates are fixed when we give specific board information to Krycek. However, if the power limitation for DNN layer is larger, there will be more configuration candidates kept after filtration process, which means that the PE array will get more chance to find out better configuration for every DNN layer. Also, the size of the PE array could affect the improvement increase. The PE array with larger size will have more resource to cover the DNN network and generate more configuration candidates. So in the Figure 4.26 and 4.25, when the power constraint increases a little bit, improvement of energy and performance will increase and the PE array with large size will have more massive improvement compared with smaller size PE array.

However, if the power constraint is large enough, most of the configuration candidates will be kept. And then the improvement will gradually stay the same. In other words, if the power constraint arrives at the threshold of the PE array, the improvement's value will not increase anymore.

Chapter 5

Conclusion

We notice that DNN has been used in a lot of areas and can change human's life. To make the DNN's performance and energy better is eagerly needed. A series DNN accelerators have been invented, which succeed mapping DNN on the hardware board rather than on generic processor. In this thesis, we propose that if we can make the hardware board dynamic reconfigurable for different DNN layers' execution instead of applying static configuration for every layer, the performance and energy metrics can be largely improved.

To verify this possibility, we need to generate the configuration candidates first, which might be implemented on the hardware board. The Krycek is a kind of configuration generator. Also, it can evaluate the configurations and filter out the unqualified configurations. In other words, Krycek first receives hardware information such as L1 size, FIFO bandwidth, etc. Then it will produce all configuration under the requirement of hardware resources. After these steps, the Krycek can store these configuration candidates into a file. When the users pass constraints like power and performance to Krycek, it will call the file to find out the best configuration.

For DNN's reconfiguration, it starts with a basic static configuration, which is suitable for every DNN layer. Then, for the next layer, it will try to find whether there is a better configuration. If the better configuration exists, the Krycek will judge whether it's worth doing reconfiguring. Because the cost for reconfiguration should be taken into consideration, Krycek will compare the overhead cost of reconfiguration to keeping the previous configuration before applying configuration or not.

The Chapter 'Reconfigurable DNN accelerator' shows the results of improvement of different DNNs when the dynamic reconfiguration is applied. From the result Figure 4.9 and Figure 4.8, we can observe about 2x improvement of energy and performance averagely between several DNNs. Also, the Figure 4.7 shows the tendency of performance improvement when sweeping the power constraint.

The Krycek gives the software part evaluation, and then the CNN-SAC is going to verify it

on the hardware part. The CNN-SAC can handle with DNN computation in systolic array architecture. Also, it can dynamically change the PE array's shape by passing different control signals to multiplexers. We can get the configuration list from Krycek, and then we translate these configurations into control signals. So for every DNN layers, we will apply different controls signals. The Figure 4.25 and Figure 4.26 shows the energy and performance improvement of different PE array size under distinct power constraints. From this figure, we can notice that the higher power constraint will bring up higher improvement before power constraint arrives the threshold. And the average improvement is about two times versus static configuration.

Chapter 6

Future Work

There are two major parts that we are going to continue working on. The first one is about the Krycek, to make the Krycek more flexible. Currently, Krycek is just focusing on the popular neural network. However, it will be excited if we can implement any custom neural network on Krycek, which means that users just need to give the network's information and then Krycek can generate the configuration candidates accordingly. Also, we are going to add more features to Krycek and to make the evaluation of Krycek more accurate. The second part is that we still can improve our CNN-SAC. Because we just make reconfiguration through changing the weight data source. We could also change the input data source to make the reconfigurability more powerful. Also, the CNN-SAC will be equipped with more configurations under the constraints of physical design.

Through improving these two parts, the whole design will be more flexible and accessible. So any DNNs can be implemented and evaluated first and best configurations list can be found in Krycek. Then, we can achieve the better performance and power efficiency after apply this configurations in our CNN-SAC system when running the DNN executions.

Bibliography

[1] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin, 2015.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[3] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[4] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 27–39. IEEE Press, 2016.

[7] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[8] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops 2011)*, pages 109–116. IEEE, 2011.

[9] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[14] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[15] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19. IEEE, 2013.

[16] Mateja Putic, Swagath Venkataramani, Schuyler Eldridge, Alper Buyuktosunoglu, Pradip Bose, and Mircea Stan. Dyhard-dnn: Even more dnn acceleration with dynamic hardware reconfiguration. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 14:1–14:6, New York, NY, USA, 2018. ACM.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[18] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, 2004.

[19] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[20] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[21] T. Tanigawa and K. Kamijo. Stock price pattern matching system-dynamic programming neural networks approach. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 2, pages 465–471 vol.2, June 1992.

[22] Yibo Wang and Wei Xu. Leveraging deep learning with lda-based text analytics to detect automobile insurance fraud. *Decision Support Systems*, 105:87–95, 2018.