# Subtractor-Based CNN Inference Accelerator

by

Xiaohang Gao

Submitted in partial fulfillment of the requirements

for the degree of Master of Applied Science

at

Dalhousie University

Halifax, Nova Scotia

October 2022

# DEDICATION PAGE

To my mother, Guangfang Suo, for her unconditional love and support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABSTRACT

In this thesis, a novel method to boost the performance of CNN inference accelerators by utilizing subtractors has been proposed. After analyzing the distribution of the weight, the distribution characteristic was exploited to develop this method. The proposed CNN preprocessing accelerator relies on sorting, grouping, and rounding the weights in order to create combinations that allow for the replacement of one multiplication operation and addition operation by a single subtraction operation. Given the high cost of multiplication in terms of power and area, replacing it with subtraction allows for a performance boost by reducing the power and area. The proposed method allows for controlling the trade-off between the performance gains and the accuracy loss through increasing or decreasing the usage of subtractors. Using a rounding size of 0.05 on LeNet-5 with the MNIST dataset, the proposed design can achieve 32.03% power savings and a 24.59% reduction in the area at the cost of only 0.1% in terms of accuracy loss.

# LIST OF ABBREVIATIONS USED

| | |
|---|---|
| ALU | Arithmetic Logic Units |
| AU | Activation Unit |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| DBB | Density-Bound Block |
| DSM | Dynamic Segment Method |
| GeMM | General Matrix Multiply |
| GPU | Graphics Processing Unit |
| HBM | High-bandwidth Memory |
| IoT | Internet of Things |
| LBCNN | Local Binary Convolutional Neural Network |
| MAC | Multiply-accumulator |
| MLP | Multilayer Perceptron |
| MXU | Matrix Multiplier Unit |
| NNZ | Number of None-Zero |
| PE | Processing Element |
| SA | Systolic Array |

| | |
|---|---|
| SSM | Static Segment Method |
| STA | Systolic Tensor Array |
| TPU | Tensor Processing Unit |

# ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my supervisor and mentor, Dr. Kamal El-Sankary, for the continuous support of my master's research, and for his patience, enthusiasm, and immense knowledge. He always had ideas and valuable suggestions that helped me overcome my research difficulties; without his help, I could not have come up with this idea and finished this research. I could not have imagined having a better mentor for my master's study.

Besides my mentor, I would like to thank my co-supervisor, Dr. Issam Hammad, who gave me my first machine learning course and provided selfless help on my paper. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. I would like to thank the rest of my supervisory committee members, Dr. Jason Gu and Dr. Guy Kember. I would also like to thank department administrator Tamara Cantrill and Secretary Nicole Smith for their help.

Last but not least, I would like to thank my parents, Guangfang Suo and Wenlong Gao, for supporting me throughout my life, my fiancée Yang Wang for her trust and care, and my other family members' care and understanding.

# Chapter 1 INTRODUCTION

## 1.1 Motivation

Deep learning techniques using convolutional neural networks are prevalent artificial neural networks, which are widely used and demonstrate effectiveness for various application scenarios. The Convolutional Neural Network (CNN) has existed for over two decades. When CNN is compared with other neural networks, for example, multiple layer perceptron (MLP), it uses kernel as a character extractor to process the images. In some of these domains, CNNs achieve better accuracy than humans [1-3]. CNNs are used in various applications, including real-time image reconstruction [4], human action recognition [5], brain tumor detection [6], and building structural damage recognition [7]. When applied in specific applications such as mobile devices and self-driving cars, they usually require repaid response while maintaining acceptable accuracy. The computational burden increases dramatically with the use of deeper and/or wider architectures to achieve a more outstanding prediction performance and accomplish a more difficult target. On the other hand, it also requires low or limited power consumption to maintain the target standby time. The popular implementation platforms of CNNs are Graphical Processing Units (GPU) and Tensor Processing Units (TPU). Those two hardware solutions can perform both training and inference with low latency and high throughput at the cost of elevated power consumption. However, with diverse application scenarios, especially in the extensive Internet of Things (IoT) market, the execution time and power consumption become critical issues. As a solution, the training process will be done on the cloud, and the inference part will be done on the embedded device, as shown in Fig.1.1[8]; meanwhile, various

accelerator and approximate methods also been applied to further improve the performance and reduce resource consumption.

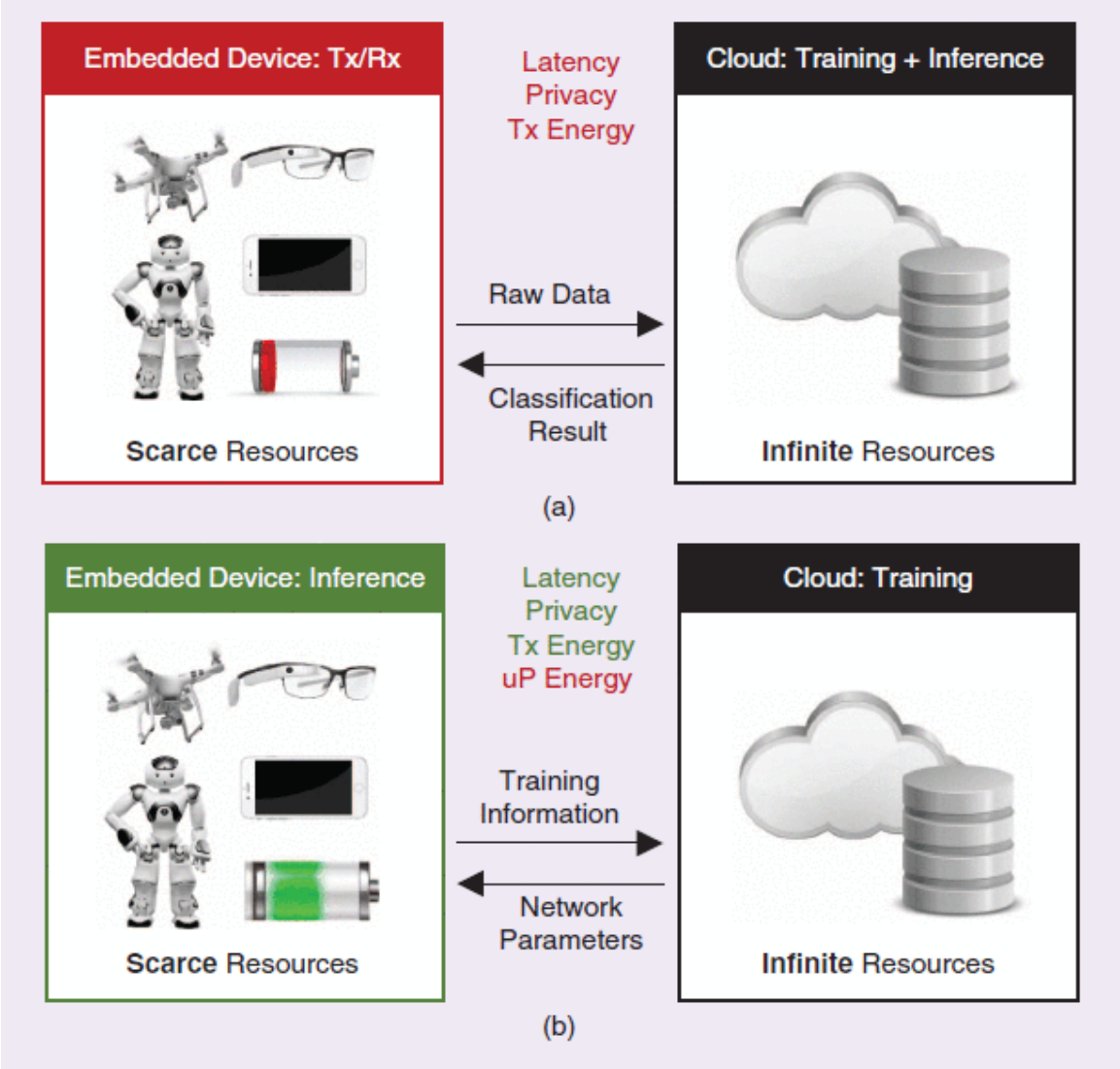


Figure 1.1 Concerns about power consumption and recognition latency where (a) all to cloud and (b) do inference at local [8]

As shown in Fig.1.1 [8], solution (a) transmits all data (image or video) to the cloud to do the training and inference, whereas method (b) is only training in the cloud, and inference is at the local. Thus, in this thesis, we are focusing on providing a CNN

accelerator by preprocessing the network parameters in Fig.1.1[8] at the cloud and creating a corresponding modified version of convolutional operation at inference (local) to improve the performance of power and area savings.

## 1.2 Contributions

In this thesis, a Subtractor-Based CNN Inference Accelerator is presented. The accelerator is divided into two parts, the weights preprocessor and the modified convolution unit; additionally, performance analysis has also been provided. In Summary:

1. A weights preprocessor is presented in this thesis. The preprocessing works before the inference. When receiving a trained model, it reads the values of the weights, exploits the characteristic of weights distribution to sort, approximate and generates new approximate weights to replace the original weights in the trained model.

2. A modified convolution unit is implemented; it works at the inference level, reads, and processes the special weights combinations by adding a feature to the original convolution operation.

3. Synopsys Design Compiler made the power consumption test with TSMC 65 nm technology. The software implementation was tested on the Google Colab platform with Pytorch.

## 1.3 Thesis Organization

The thesis is organized as follows:

1. Chapter 2 provides a background of CNN and literature reviews of the peers' works in related approximate computing accelerator design.
2. Chapter 3 presents the design of our approximate accelerator
3. Chapter 4 presents the details of the implementation, including the codes.
4. Chapter 5 conclude the thesis and discusses future works.

# Chapter 2 LITERATURE REVIEW

## 2.1 LeNet-5 and MNIST database

In this thesis, the LeNet-5 [9], an image recognition application that categorizes handwritten digits, was chosen to test, modify and perform the simulations. The simple structure of this model is as shown in Fig.2.1, the corresponding dataset is the MNIST database [10], and the samples are shown in Fig.2.2.

The network is termed LeNet-5 since it contains five layers with learnable parameters. It has three sets of convolution layers with an average pooling combination. We have two fully connected layers after the convolution and average pooling layers. Finally, a Softmax classifier classifies the objects into their corresponding category. The input to this system is a one-channel 32x32 grayscale image. Convolution, to extract the characteristic of the target image, for example, the first convolution operation, applies the first convolution operation with a six-channel 5x5 filter to the input; it will create a six-channel 28x28 feature map. Subsampling, also called the pooling layer, is a technique to reduce the reliance on precise positioning within feature maps that takes results from a convolutional layer, a 6-channel 28x28 feature map, through a 2x2 pooling layer. After two more convolutional layers and one more subsampling, it goes to a fully connected layer. Notice that the MNIST database images are 784 pixels or 28x28, which is different from the input of LeNet-5; thus, to get the MNIST images dimension meets the requirements of

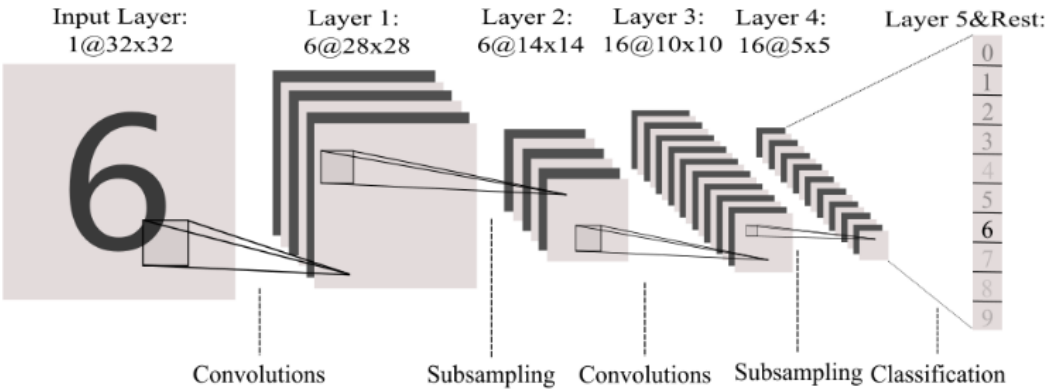


Figure 2.1 Convolutions and subsamplings of LeNet-5

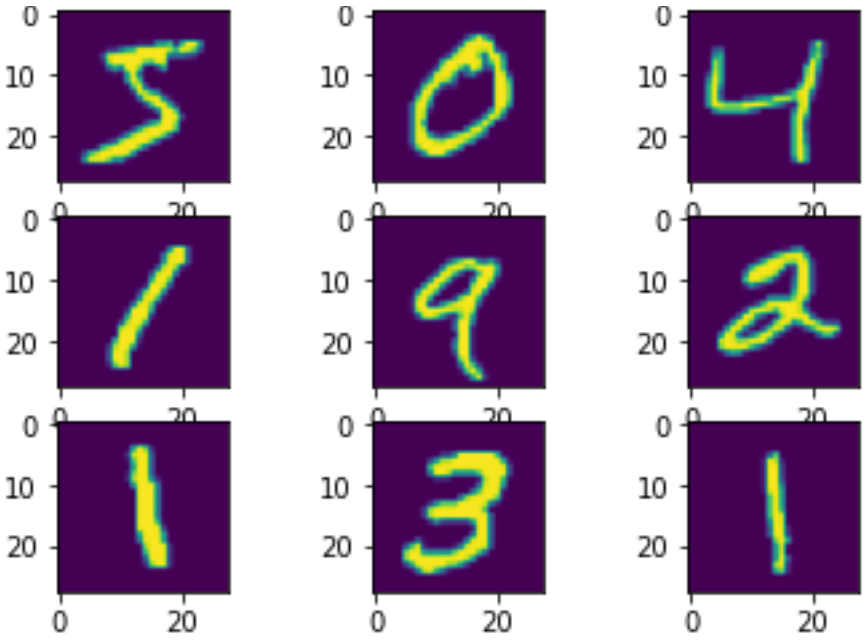

Figure 2.2 Samples in MNIST datasets

the input layer, the images from MNIST will be padded to 32x32, where the padding is to add zeros to extend the original image size.

The focus of this work is investigating the convolutional layers. Each convolutional layer keeps multiplying and adding, called the multiply-accumulate (MAC) operation. The input and output for the first convolutional layer are 32x32x1 (size of kernel x size of kernel

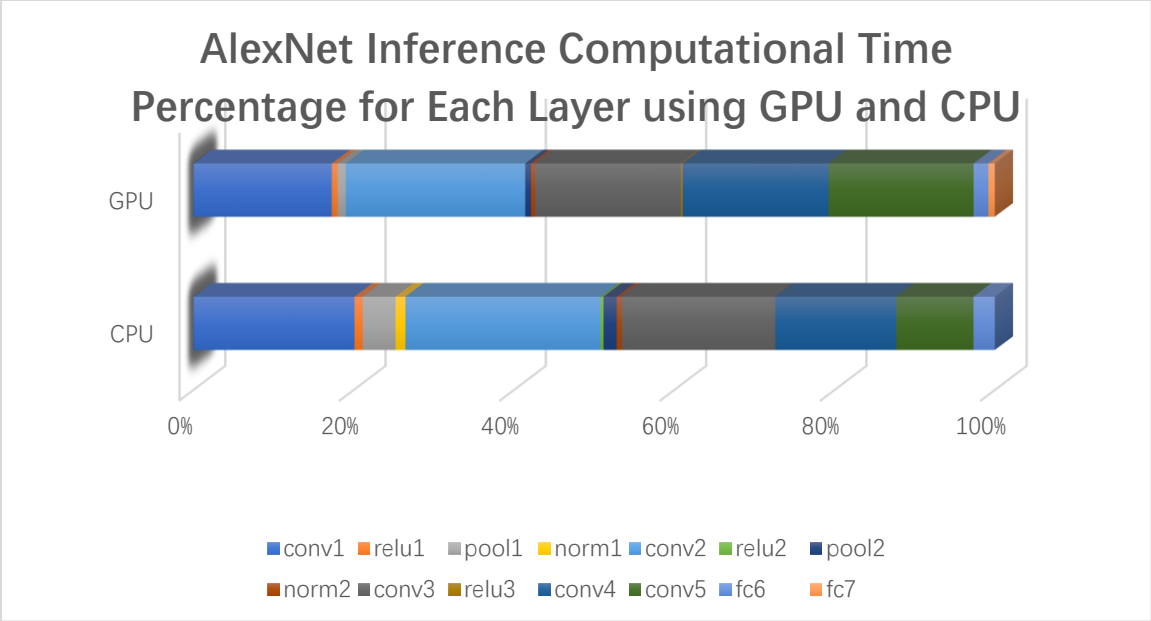


Figure 2.3 AlexNet inference computational time percentage for each layer using GPU and CPU

x number of the channel) and 28x28x6, respectively, which will perform 117,600 (28x28x5x5x5) multiplications and additions. In second and third convolutional layers are 240,000 and 48,000, respectively. In three convolutional layers, it takes 405,600 multiplications and additions in total. The research in [11] shows that the convolutional Flayer consumes the 90%-time execution time, as shown in Fig.2.3.

The fully connected layer has three sets, from 120 to 84 to 10. The reason for ten nodes at the end is LeNet-5 is to recognize handwriting digits, where the numbers are from 0 to 9; thus, the ten nodes at the end with the highest value will be the answer generated by the neural network.

Achieving a more accurate prediction in CNN requires higher computational energy as deeper and wider architectures are utilized. Computation accuracy can be adequately tuned to the specific application requirements to reduce power consumption.

Hence, introducing methods that can reduce the CNN computation complexity and therefore reduce the overall required energy is needed. Previously, several methods have been published which aim to reduce the CNN computation complexity; this includes parameters pruning [12-14], binary neural network [15-20], sparsity [21], and approximate multipliers [22-28]. Neural network pruning is a simple yet effective strategy for deleting unnecessary synapses and neurons to lower the model's size, in other words, to generate sparse neural networks. Binary neural networks, as a quantization method, BC [15] as the simplified version, simplifies the original complex convolutional operations where the weights only contain -1 and 1, while others may have more values. Approximate computing is used for such media-related systems due to their ability to tolerate error; one of the representative approximate computing is the approximate multiplier, such as stochastic computing [28], which reduces power, area, and delay where the cost is the inaccuracy. In the next section, we will pick several classic related methods to analyze in depth.

## 2.2 Approximate Multipliers

In [23], a flexible use of the approximate multiplier method was proposed. It first reviewed two popular approximate multipliers, the static segment method (SSM) and the dynamic segment method (DSM); those two methods accelerate CNN by manipulating the data structure. Later it came up with a preprocessing precision controller on inference to detect and predict the adequate approximate precision to choose the best segment sizes for the approximate multipliers.

SSM, as its name, chooses a fixed length for segmentation. An n-bit integer number will be divided into static m-bits segments with k offset between the initial bits of two

consecutive segments. An example of SSM for a 16-bit integer number with k=4 and two different segment sizes where m=4 and m=8 can be found in Fig. 2.4. Notice that when using SSM, the start point of each 16-bit integer number is fixed, it depends on the value of k, and the length is determined by the value of m. The sign bit in two numbers will be extracted to calculate the final sign, and the original position of the sign bit will be replaced as zero when performing segmentation.

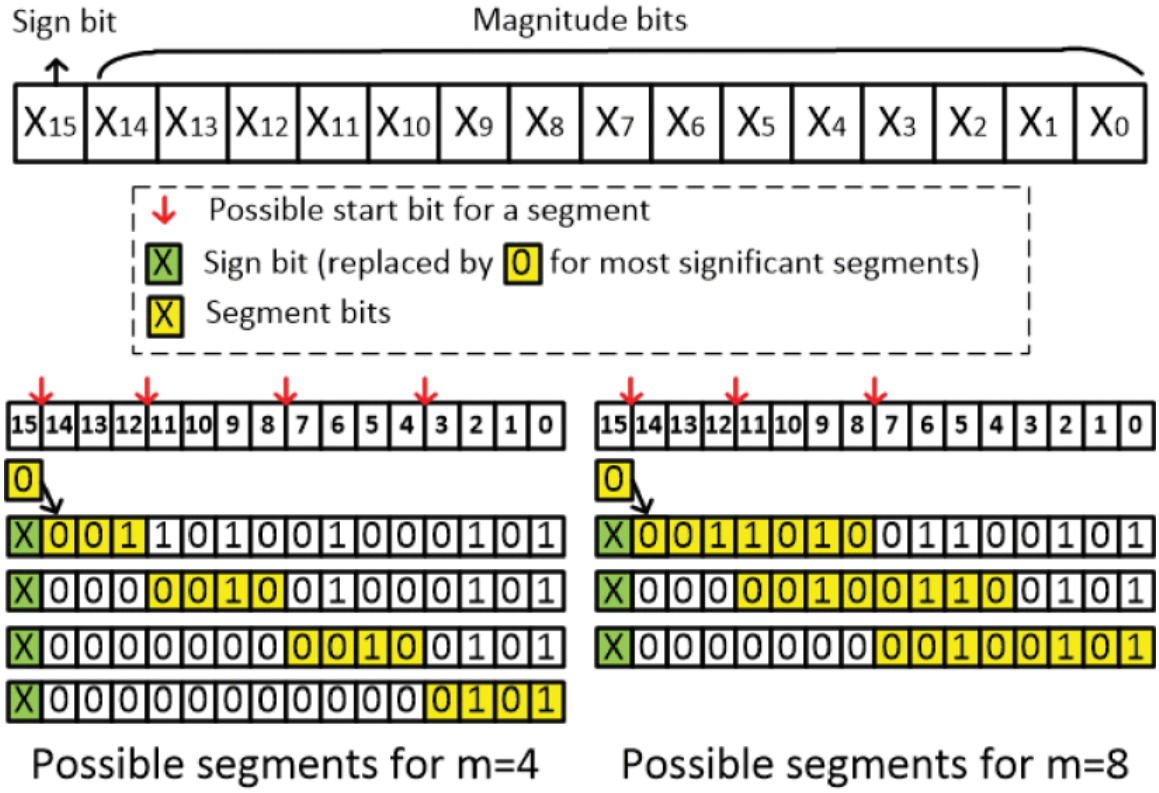


Figure 2.4 SSM for K=4 with m=4 and m=8.[23]

Compared with SSM, the DSM has different logic in segmentation which is more complicated and costly. The DSM method will try to find the first non-zero bit except the sign bit, and then it extracts the following m-1 bits to perform multiplication. A close version of DSM is DRUM, which will set the last extracted bit to one, as shown in Fig.2.5.

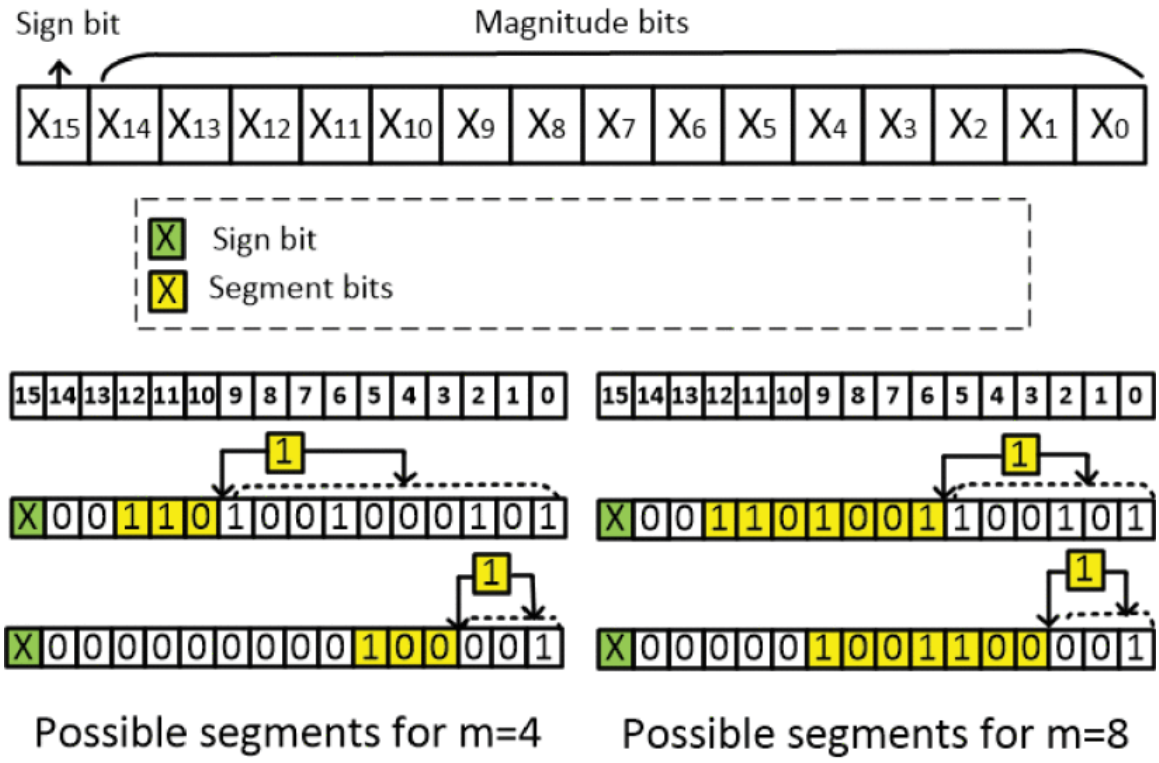


Figure 2.5 DSM for K=4 with m=4 and m=8. [23]

Based on their studies, they found that after applying the approximate multipliers, the CNN's inference accuracy varies widely between image classes; certain image classes may achieve even higher accuracy using low-precision approximate multipliers compared with high-precision approximate multipliers, and for some other image classes, there is no difference in CNN's inference accuracy between low and high precision approximate multipliers. Thus, they create a small neural network that contains three 2D convolution layers to study this phenomenon. In the end, whenever a set of image batches tries classification, they will first go through this small CNN to predict the best proper choice of K and M to get the best precision; this process is shown in Fig 2.6.

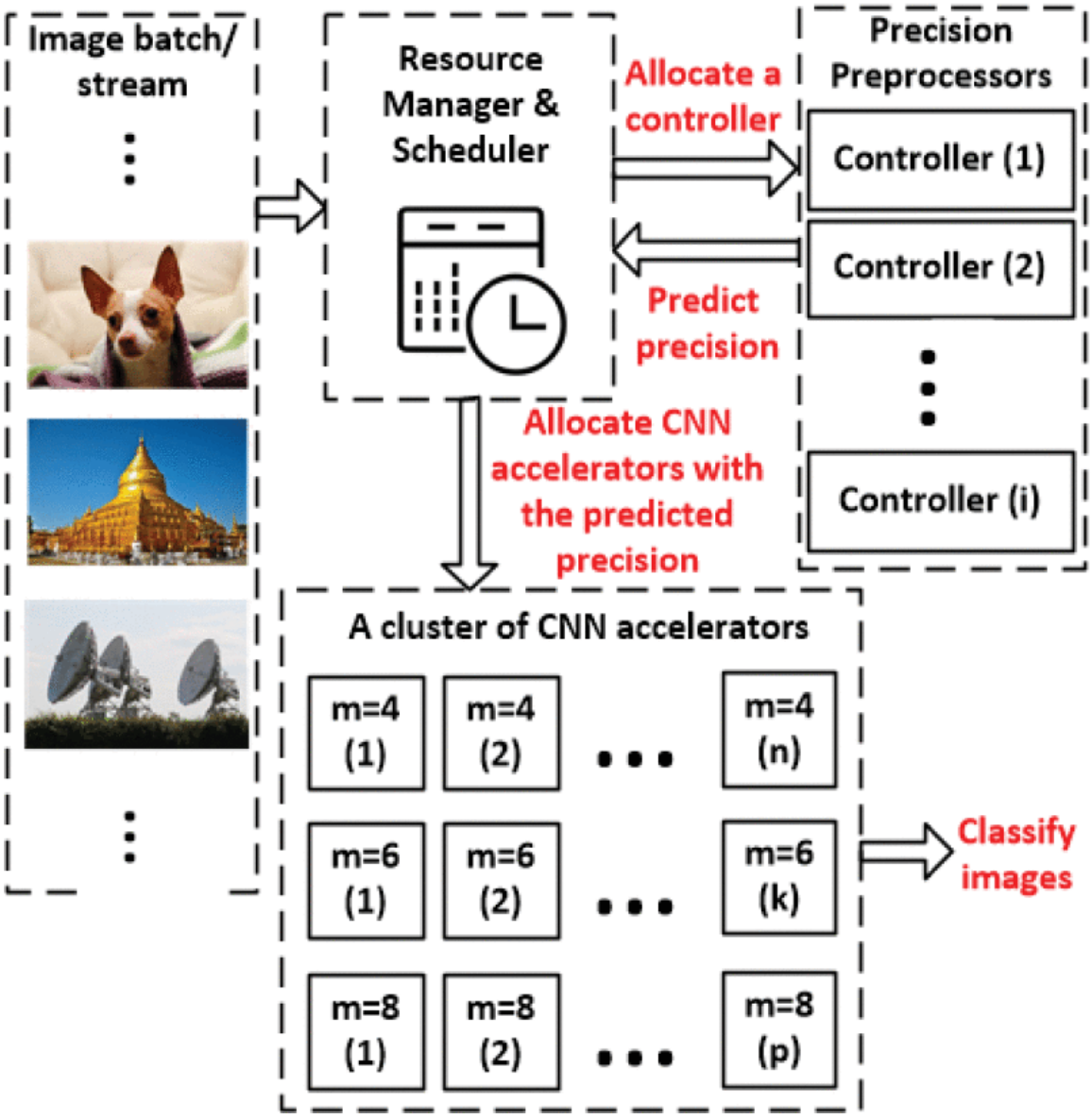


Figure 2.6 Preprocess the images with precision controller CNN [23]

## 2.3 Binary Convolutional Neural Network

LBCNN [17] is CNNs with LBC layers. The LBC layers will approximate the non-linearly activated response of standard convolutional layers. The module of LBCNN, as shown in Fig.2.7, replaces the original weight matrixes with binary matrixes. As we can see that the original response map was replaced by a difference map, and a bit map was inserted between the difference map and the feature map. The difference map is generated by the input image with LBC filters, where the LBC filters are predefined fixed convolutional filters. Then, the difference map goes through a non-linear activation function to produce the bit maps. Finally, the LBC layer response is generated by the bit maps with learnable weights.

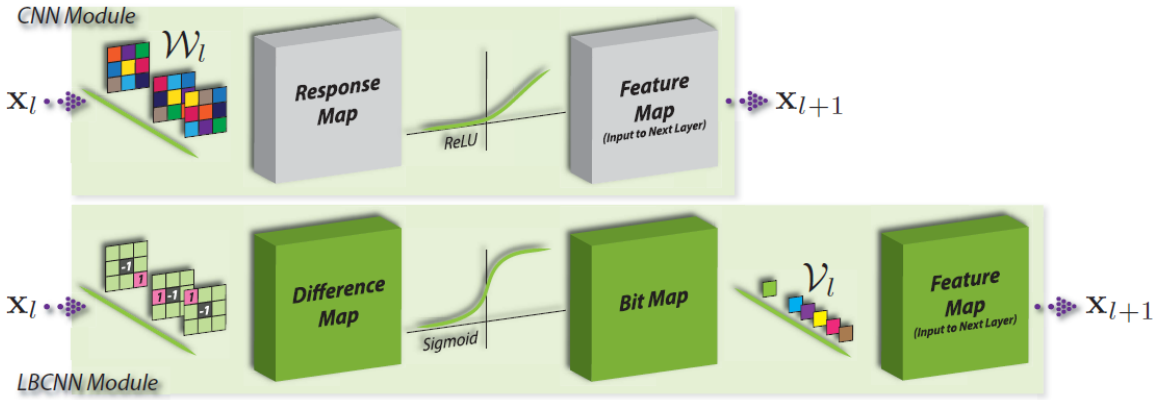


Figure 2.7 Module comparison between CNN and LBCNN [17]

In [16], an accelerated platform was proposed to reduce MAC operations and improve overall performance. As mentioned in LBCNN, the weights are changed to binary numbers; thus, many zeros will exist in weights. This method exploits this particular structure to take advantage of sparsity to reduce memory access and total MAC operations. To achieve that, [16] consists of two modules, as shown in Fig.2.8 it has a weight preprocessor and a layer accelerator.

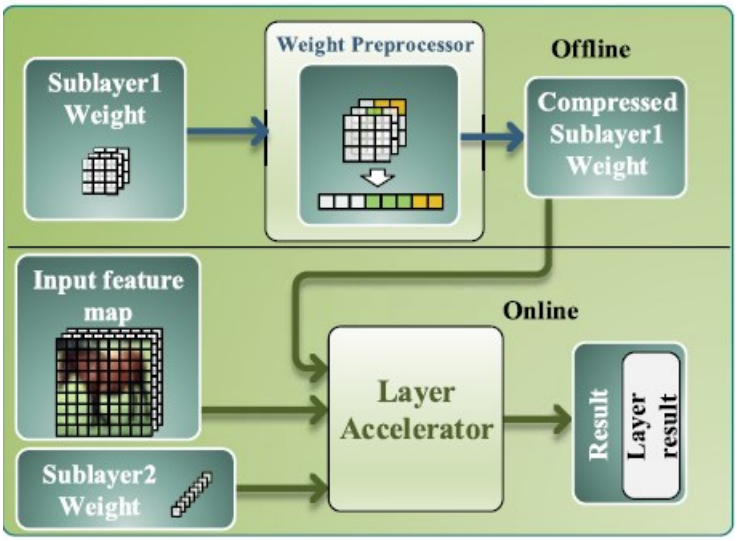


Figure 2.8 Platform overview [16]

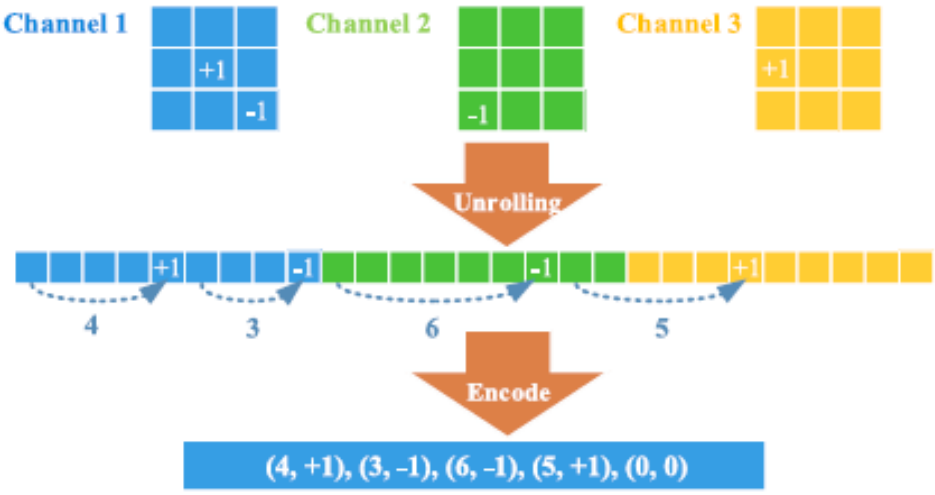


Figure 2.9 An example of weight preprocessing [16]

The first part of this method is a weights preprocessor or a weight encoder. As shown in Fig. 2.9, the matrixes first unrolling to an array, based on the position of +1 and -1, it extracts the distance information and starting position. In that array, the distance between the first non-zero bit is 4, and that bit has a value of +1, which turns into (4, +1). Following this logic, the original array turns into (4, +1), (3, -1), (6, -1), (5, +1), (0, 0). The original matrix has 3 x 9 = 27 values, after the weight preprocessor, it only has 2 x 5 = 10

values, it saves 27 – 10 = 17 values.

Since the Data structure was changed in weight preprocessing, it came up with a layer accelerator was used to extract the corresponding values in the input feature map, as shown in Fig.10.

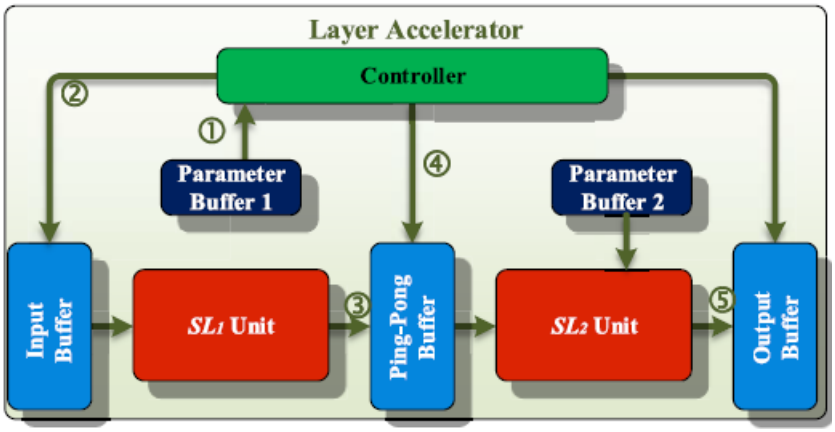


Figure 2.10 Hardware design of the layer accelerator [16]

## 2.4 Sparse Analysis in CNNs

In [21], two improvements to the traditional Systolic Array (SA) were proposed. The first improvement is to bring Tensor into SA to create a new microarchitecture called Systolic Tensor Array (STA). The STA increases intro-PE operand reuse and data path efficiency. Second, they used a new data format called Density-Bound Block (DBB) to reduce the total number of operations.

The neural networks highly rely on large-scale matrix processors. The conventional CPUs are good at quick prototyping and require maximum flexibility, and miniature models have small batch sizes and do not need a long time to train; GPUs are good at medium size models; TPUs is targeting the extremely large models that need to train for

weeks or months, and the heart of the TPUs are SAs [29].

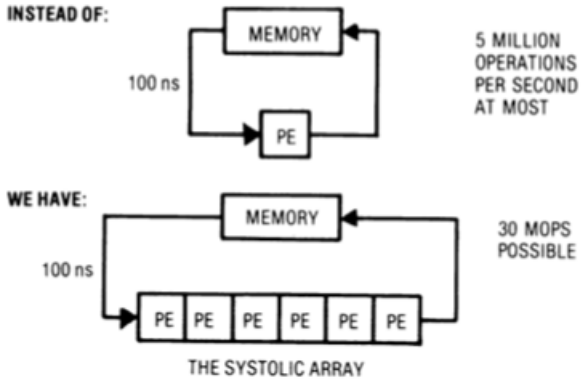


Figure 2.11 Basic principle of a systolic architecture [30]

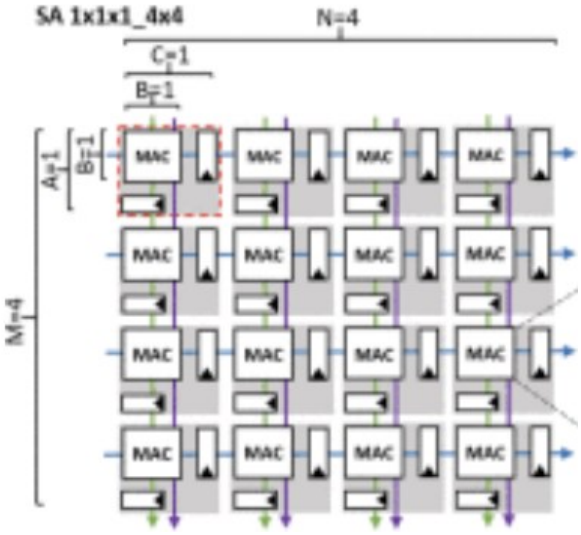


Figure 2.12 Conventional systolic array ABCMN=1,1,1,1,4,4 [21]

The basic principle of a systolic array is shown in Fig. 2.11 [30]. The advantage of this architecture is the Processing Elements (PEs), which are Multiply-Accumulators (MAC), are connected directly to each other to form a large physical matrix, as shown in Fig. 2.12 [21].

In the SA structure, as each multiplication is executed, the result will pass to the next directly connected MAC as the green line, and the output results are the accumulation between each multiplication.

In [21], they fuse blocks of scalar PEs into a single tensor PE, called Systolic Tensor Array (STA), as shown in Fig. 2.13. The STA in this example contains M x N tensor PEs, for each PE includes A x C MACs performing the dot-product operation on B operand pairs. Compared with conventional SA, the STA reduces the number of operand buffers per MAC by 2x and the number of accumulator buffers per MAC by 4x.

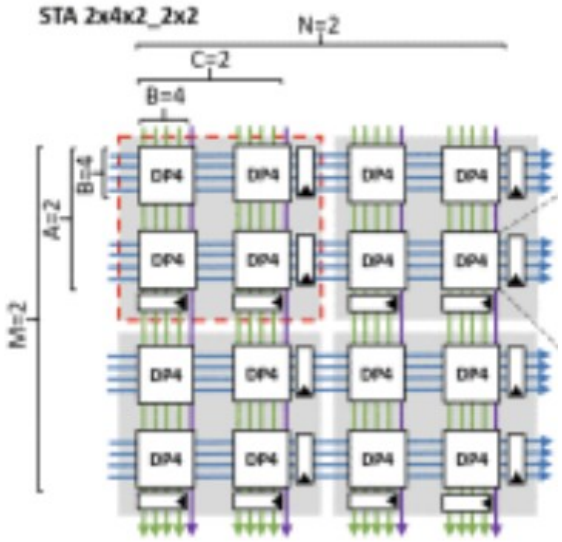


Figure 2.13 Systolic tensor array ABCMN=2,4,2,2,2 [21]

The second contribution in [21] is to create a new sparse matrix, as shown in Fig. 2.14 (c). CNNs' layers have sparsity in the weight data and activation data; zero in data can be exploited to skip the corresponding MAC operation, which can improve the performance of throughput and power consumption. In Fig. 2.14, (a) represents random

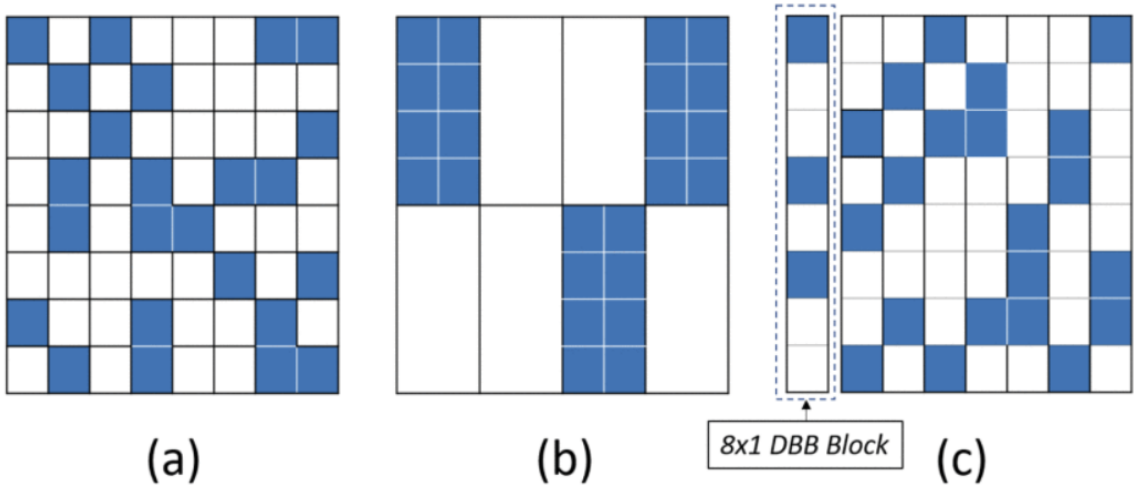


Figure 2.14 Sparse matrices: (a) random sparse; (b) 4x2 block sparse; (c) proposed 8x1 density bound block (DBB). Blue square represents number of None-Zero (NNZ) [21]

sparse which is very difficult to exploit; (b) is block sparse, it groups zero weights into coarse-grained blocks, which results in low accuracy; (c) the proposed DBB, it sets an upper limit to the NNZ for each block, comparing with (a) and (b), the proposed method is a middle-ground sparse format. DBB can achieve better accuracy performance compared to block sparse because the distribution of NNZ is less constrained.

They made tests on DBB and STA separately to evaluate the feasibility of each model. The first demonstration is for DBB with 8-Bit data under various popular CNN models, including VGG-16, MobileNetV1, ResNet-50V1, ConvNet, and LeNet-5 on ImageNet, CIFAAR10, and MNIST datasets, as shown in Table. 2.1 [21].

After adding support to STA for DBB, under the situation of NNZ smaller or equal to 4, the STA-DBB can reduce 50 percent MAC operations, where each 8-input PE only requires 4 MACs rather than 8. It can not only save the clock cycles but also save the area and power. After testing different array sizes and comparing the best performance of

different designs, the power and area efficiency results for comparison between STA-DBB and other techniques are shown in Table. 2.2.

Table 2.1 CNNs training with 8-Bit length weights of DBB [21]

| **Model** | **Dataset** | **Baseline Acc. (%)** | **Pruned Acc. (%)** | **Model Result NNZ (%)** |
|---|---|---|---|---|
| **LeNet-5 (DBB)** | MNIST | 99.1 | 98.7 | 1.05K (25) |
| **ConvNet (DBB)** | CIFAR10 | 86.0 | 85.3 | 26.8K (25 |
| **MobileNetV1(DBB)** | ImageNet | 70.9 | 69.8 | 1.6M (50) |
| **ResNet-50V1(DBB)** | ImageNet | 75.2 | 74.2 | 8.79M (37.5) |
| **VGG-16 (DBB)** | ImageNet | 71.5 | 71.4 | 5.39M (37.5) |
| **AlexNet** | ImageNet | 57.4 | 57.5 | 2.81M (75) |
| **VGG-19** | ImageNet | - | 71.8 | 4.1M (12.6) |

Table 2.2 Throughput-Normalized area and power efficiency with 50 percent DBB at 1GHz [21]

| **Design** | **Model Sparsity** | **Array** | **Area Eff** | **Power Eff** |
|---|---|---|---|---|
| **SA-NGG** | Dense | 1x1x1 | 0.95 | 0.65 |
| **SA** | Dense | 1x1x1 | 1.00 | 1.00 |
| **STA** | Dense | 4x8x4 | 2.08 | 1.36 |
| **SMT-SA** | Random (62.5%) | T2Q4 | 1.21 | 0.80 |
| **STA-DBB** | DBB (50%) | 4x8x4 | 3.14 | 1.97 |

## 2.5 Conclusion

At the beginning of the chapter, we first introduced the background of LetNet-5 and its training database, MNIST. The time distribution of the CNN has also been reported, it shows the convolutional layer consumed over 90% of the total execution time, and the Convolutional layer mainly consists of MAC operation. Thus, the convolutional layer is a great point to start with to accelerate neural networks.

The rest sections covered related works in a different approach. The common factor among those methods is manipulating the weights data structure.

In 2.2, it uses the proposed approximate multiplier to reduce the total operations by discarding certain values to achieve approximate computing. Later, it came up with a precision preprocessor to predict precision. This section provides ideas for the manipulation of weights and preprocessing.

In 2.3, it has a similar architecture to 2.2, which also contains a preprocessor. The difference is the preprocessor in 2.3 is targeting weights and has a corresponding MAC unit proposed to support the preprocessed weights' structure. By exploiting the characteristic of weights in BCNN, the preprocessor converts the original weights to a special data structure: distance + value.

In 2.4, it reduces the operations by exploiting sparse in weights and introduces a new sparse block called DBB to improve further the performance of reducing MAC operations requirements and the efficiency of power and area.

# Chapter 3

# PROPOSED ACCELERATOR

This chapter describes the proposed accelerator. Firstly, the chapter analyzes weights distribution for various convolutional neural networks. The analysis shows that the distribution behavior can be exploited to accelerate the CNN at the inference stage. Secondly, the chapter provides the new accelerator design details, which include two parts: the weight preprocessor to preprocess the weight from a trained model; the corresponding modified convolutional unit that handles the manipulated weights. Thirdly, the efficiencies of the proposed accelerator design were evaluated in both software and hardware.

## 3.1 Weight distribution analysis

Recall the LBCNN we reviewed in section 2.3; the author described a preprocessor based on the characteristic of LBCNN, which is that only one NNZ exists in one weight matrix. When observing the weights distributions in normal CNNs such as LeNet-5, AlexNet, and VGG-16, as shown in Fig. 3.1-3.6, the weights are spread symmetry around zero, which is a normal distribution. This symmetrical distribution allows for finding opposite pairs of weights that can be combined. Hence, the modern FPGA design for adder and subtractor are commonly integrated as one module [31]; the proposed method may exploit this property by utilizing subtractions to replace part of additions and multiplications to improve the performance in terms of area and power efficiency.

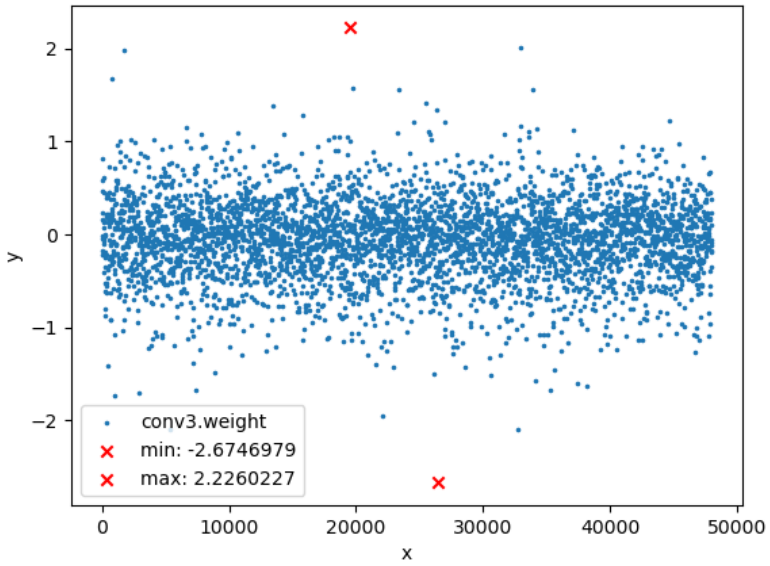


Figure 3.1 Weight distribution of layer 3 in LeNet-5

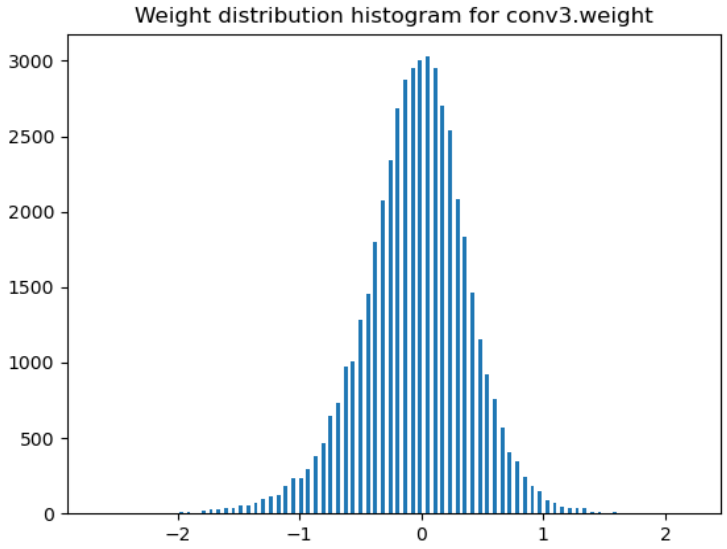


Figure 3.2 Histogram of weight distribution of layer 3 in LeNet-5

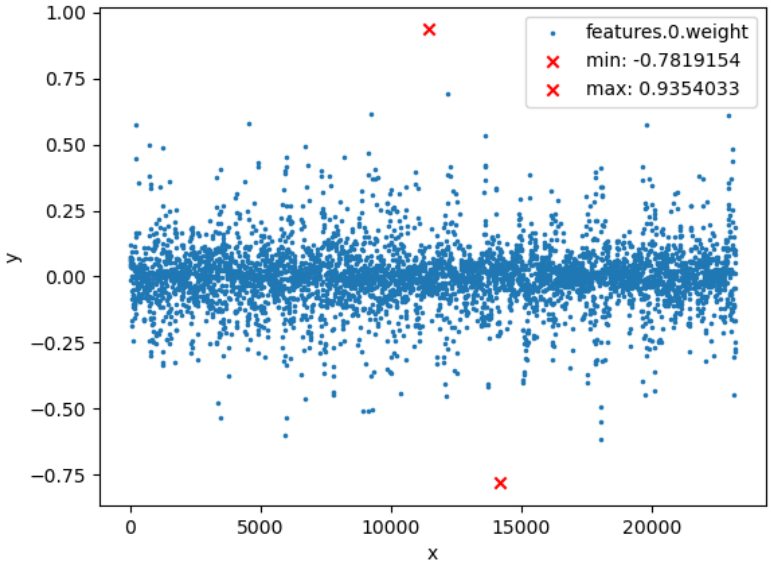


Figure 3.3 First convolutional layer of AlexNet

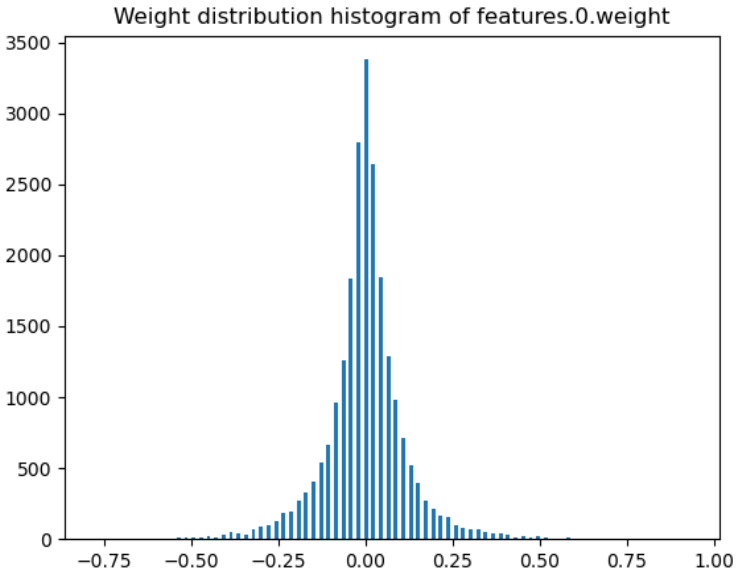


Figure 3.4 Histogram of AlexNet first convolutional layer

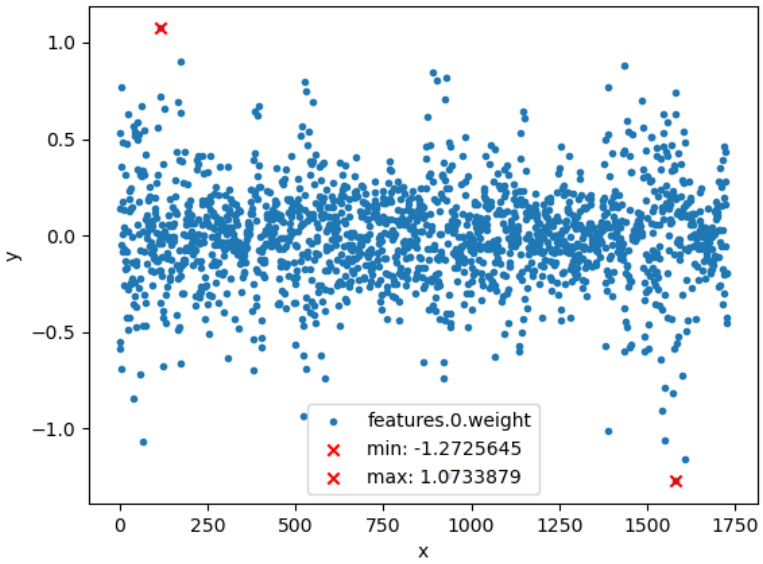


Figure 3.5 First convolutional layer of VGG-16

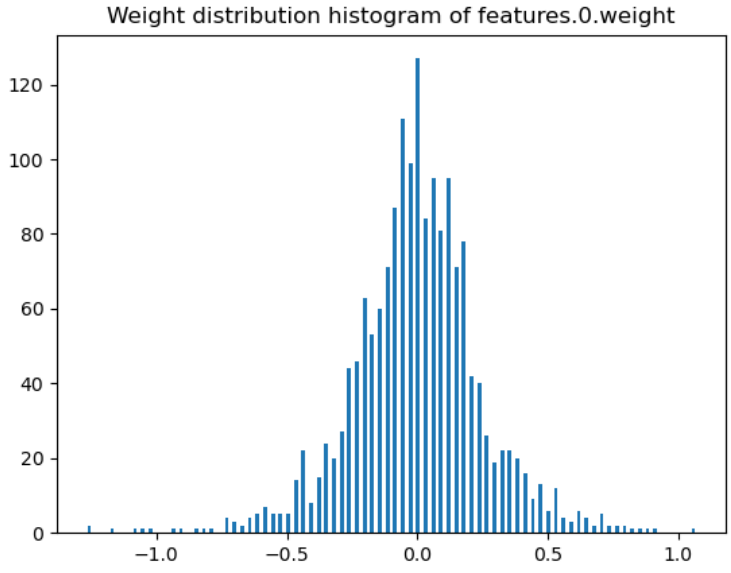


Figure 3.6 Histogram of VGG-16 First convolutional layer

## 3.2 Overview of the proposed accelerator

This section provides an overview of the proposed method, which is summarized in Fig. 3.7. The top three modules are conventional CNN's architecture at the inference level, and the bottom two modules are the proposed accelerator. The proposed method contains two blocks: a weight preprocessor and a modified convolution unit. Weights preprocessing occurs prior to the inference stage to provide the required data structure, which is used in a modified convolution unit. Recall the introduction in chapter 1; Fig.1.1 shows that modern CNNs put training at the cloud, and the user end only make inference that can save energy, reduce latency and protect user privacy. The preprocessor will be put in the cloud for further improvement in our design.

The preprocessing process relies first on sorting the weight to split original weights into positive and negative lists. Secondly, the preprocessor finds all possible combinations based on the selected rounding step and creates a list of combined weights. Finally, it splices all three lists and replaces the original weight in the CNN model.

The combined weights are executed separately at the inference level in a modified convolution unit. The combined weights will use a subtraction operation to replace one addition and one multiplication; the uncombined weights will use regular addition and multiplication.

More details about the preprocessing step are presented in section 3.3, while section 3.4 offers more information about the modified convolution unit.

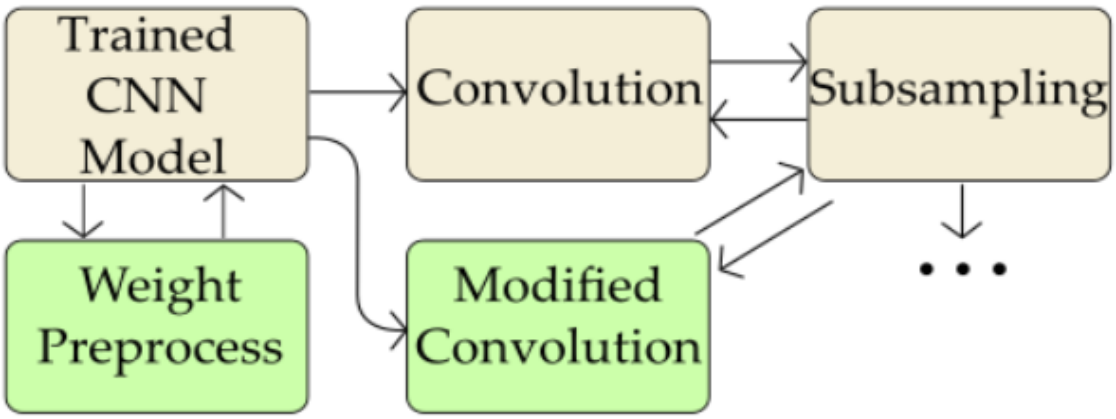


Figure 3.7 Structure of the proposed accelerator (Green)

## 3.3 Preprocessing of the Weights by Sorting and Approximation.

In this process, the original weights will go through several steps, which include sortation, finding the combination, and merging, as shown in Fig.3.8. Initially, weights will be sorted in ascending order and then split into two lists, positive and negative lists. Since the weight will move around, and at the end, weights still need to find their corresponding inputs; thus, to save the original position information during the sorting process, the argsort() function in numpy[32] is used as it holds the original position information. The argsort() function returns the index of the list while sorting. After splitting and sorting, a new flag will be inserted to indicate the current weight status. In Fig. 3.8, before finding the combination, the flag initializes to 'U', which represents Unknown status; later, based on the results of finding the combination, it will change to 'N' or 'C', which means combined or not combined.

After sorting, the next step is to find combinations of weights. As seen from the Algorithms in Fig. 3.9, weights will be combined based on a selective rounding size. A new list will be generated that contains all the combined weights which are extracted from the positive and negative weight lists, plus the rest weights in the old Positive and Negative lists; there will be three lists after finding the combination. All three lists will be merged and spliced to have all the new combined weights at the top while the rest of the weights at the bottom, as shown in Fig. 3.8.

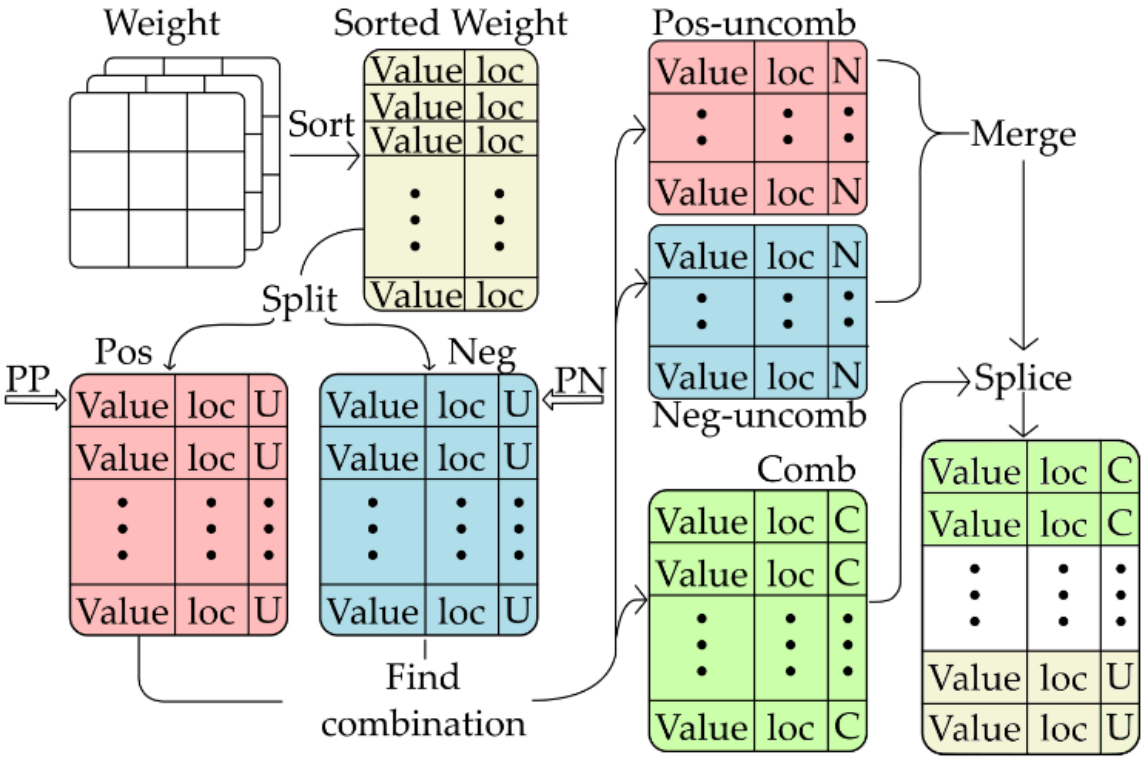


Figure 3.8 Details of weight sorting and grouping

**Algorithm 1** Find combinations

**Input:** $Pos, Neg, PP, PN, rounding$ ▷ List of sorted positive and negative weights, PP and PN are the pointers pointing to sorted positive and negative weight lists

**Output:** $Comb, Pos - uncomb, Neg - uncomb$ ▷ List of found combinations weights, and rest weights stay in original list

$idx \leftarrow 0$
$comb \leftarrow empty$ ▷ Initialize empty list
**while** $PP$ and $PN$ exists **do**
    **if** $PP.val \geq |PN.val| + rounding$ **then** ▷ Negative weights too small
        $PN.U \leftarrow N$ ▷ Assign N as no combination to current weight status
        Inc $PN$ ▷ Point to next weight
    **else if** $PP.val \leq |PN.val| - rounding$ **then**
        $PP.U \leftarrow N$
        Inc $PP$
    **else**
        $PP.U \leftarrow C$ ▷ Assign C as combination exists
        $PN.U \leftarrow C$
        $comb[idx] \leftarrow PP$ ▷ Store current weight element to comb list
        $comb[idx + 1] \leftarrow PN$
        $idx \leftarrow idx + 2$
        Delete $PN, PP$
        Inc $PP, PN$
    **end if**
**end while**

Figure 3.9 Algorithm of finding combinations

## 3.4 Combined weights input Convolution

As shown in Fig. 3.7, the convention was replaced by the modified convolution, and the input to the modified convolution was also changed to modified weights. The Modified Convolution contains two types of convolution operations; one is doing multiplication, then addition, and one is subtraction, then multiplication. There are two mechanisms in the proposed convolution unit. Firstly, determine the combined and uncombined weights. The combination of the weights which was presented in section 3.3, even though the combined and uncombined weights are merged in the same weight matrix, the data structure contains the flag U, N, C, as Unknow Non-Combined and Combined; thus, the modified convolution will include a particular convolution operation to handle the combined weights and conventional operation to execute the Unknow and Non-Combined weights. Secondly, the preprocessed weights contain another flag called Location noted as 'Loc', which is the extracted original position value generated during preprocessing to find the corresponding input. The proposed convolution operation allows for the utilization of using one subtraction as a replacement for one multiplication and one addition operation, as illustrated in equation (1)

$$I_1 \times K_a + I_2 \times K_b = K_a \times (I_1 - I_2) \quad if\ K_a = -K_b (1)$$

## 3.5 Simulation and Results

The implementation in software is based on the library provided by Pytorch [33]. In Pytorch, it has a built-in function 'conv2d' to execute the input and weight. To simulate our design, we first modify the LeNet-5 model interface, as shown in Appendix 1. code of LeNet-5 Model line 17, 20, and 23. In these three lines, we replace the original 'nn.Conv2d' with 'MYconv2d'.

Next, in function MYconv2d, we add features to execute the preprocessed weights, as shown in Appendix 2, MYconv2d. line 144-170. Line 28-79 in Appendix 2 is the sortation and find combination part; it returns the preprocessed weight data structure as the output shown in Fig. 3.8. Line 83-142 is the function viscum_V3. This function is modified based on the Einstein Summation, which is also the core function of nn.Conv2d, which performs GeMM (General Matrix Multiplication). The viscum_V3 contains both conventional and combination convolution operations and the control logic.

The hardware performance enhancements in terms of power and area were tested using a frequency of 1GHz using the Design Compiler from Synopsys with TSMC 65nm technology. All tested operations for multiplication, subtraction, and addition are followed the IEEE 758 design standard.

Fig. 3.10 and Table 3.1 illustrates the number of additions, subtractions, and multiplications using different rounding size. The table shows that by increasing the rounding size, we can increase the total number of subtractions while reducing both additions and multiplications. The total number of operations is reduced with a larger step size.

Fig.3.11 shows the relationship between rounding size, power, area, and accuracy. The left percentage is for power and area saving, and the right is for accuracy. In Fig. 3.11, the accuracy drops dramatically after step size 0.05. Thus, there is a trade-off between power, area saving, and accuracy. Using a step size of 0.05, the power can be reduced by 32.03%, while the area can be reduced by 24.59%, with an accuracy loss of only 0.1%.

Table 3.1 Number of additions, subtractions, and multiplications with different rounding sizes for LeNet-5

| Rounding Size | No. Add | No. Sub | No. Mul | Total operations |
|---|---|---|---|---|
| **0** | 405600 | 0 | 405600 | 811200 |
| **0.0001** | 399372 | 6228 | 399372 | 804972 |
| **0.005** | 313545 | 92055 | 313545 | 719145 |
| **0.01** | 288887 | 116713 | 288887 | 694487 |
| **0.015** | 276692 | 128908 | 276692 | 682292 |
| **0.02** | 265480 | 140120 | 265480 | 671080 |
| **0.025** | 259789 | 145811 | 259789 | 665389 |
| **0.05** | 242153 | 163447 | 242153 | 647753 |
| **0.1** | 233698 | 171902 | 233698 | 639298 |
| **0.15** | 228752 | 176848 | 228752 | 634352 |
| **0.2** | 225988 | 179612 | 225988 | 631588 |
| **0.25** | 223630 | 181970 | 223630 | 629230 |
| **0.3** | 222742 | 182858 | 222742 | 628342 |

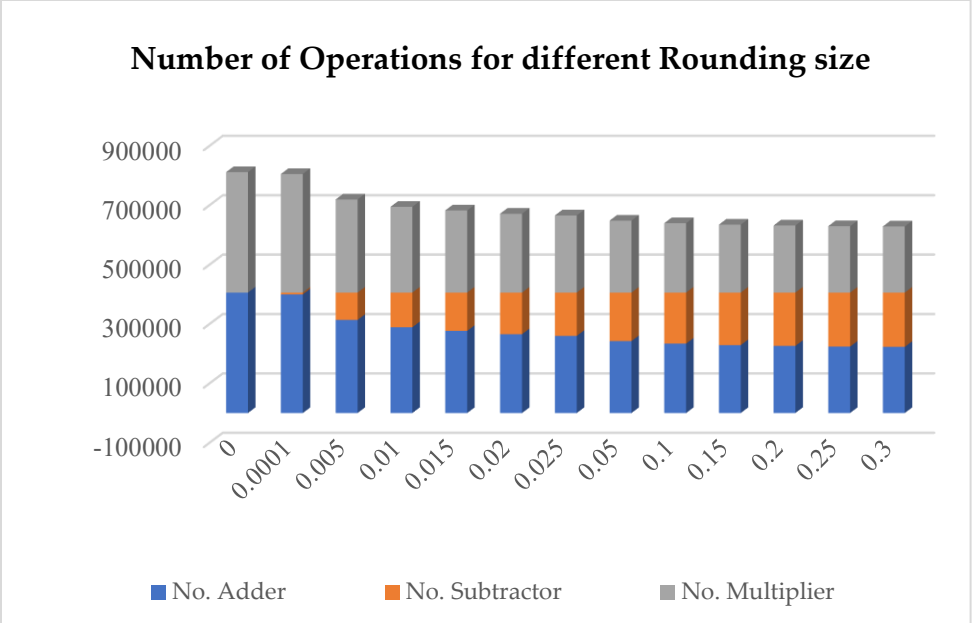


Figure 3.10 Operation portion for different rounding sizes

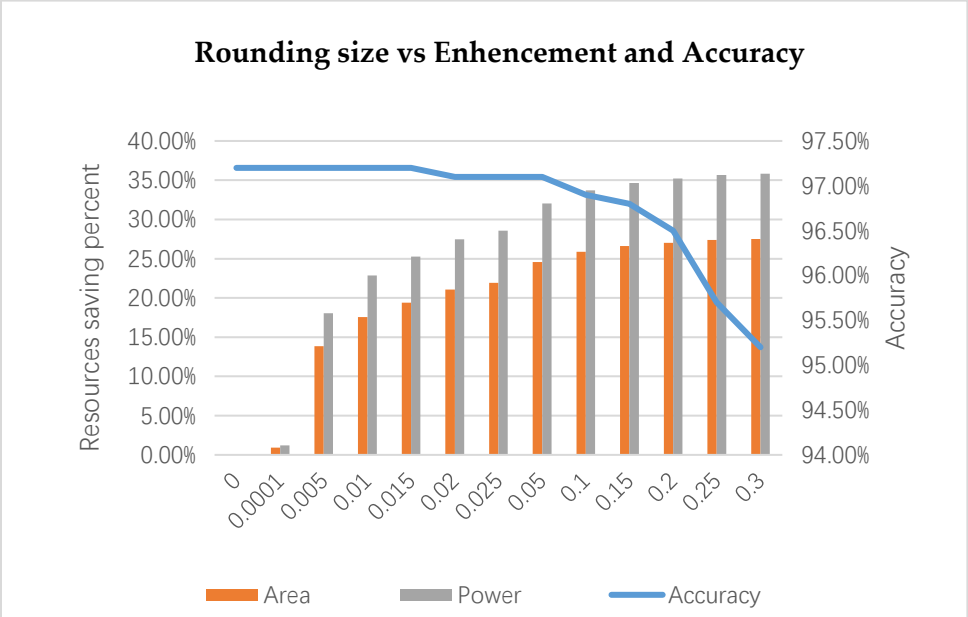


Figure 3.11 Relationship between rounding size, power, area, and accuracy performance

# Chapter 4

# CONCLUSION AND FUTURE WORK

## 4.1 Conclusion

In this thesis, we first performed an energy consumption analysis and found that the convolution operations take the most; it consumed over 90% of execution time in both CPU and GPU conditions. Secondly, by analyzing the weights distribution characteristic, we notice that the weights are normal distribution. Then, we presented a novel method for accelerating CNN that exploits our findings in weights distribution by replacing one multiplication and one addition operation with one subtraction operation. Next, we described how this could be achieved through weight sorting, approximation, and reconstruction, which allows significant performance in terms of power and area saving with minimal accuracy lost. The thesis also presented the trade-off that can be achieved between the performance enhancement and the accuracy loss based on the selected rounding size. In the end, we made tests based on CNN LeNet-5 and MNIST database. By applying our accelerator, in the case of rounding size 0.05, a power reduction of 32.03% and an area reduction of 24.59% can be achieved with only 0.1 % accuracy loss. The design allows for adjusting the trade-off between gained performance enhancements and the cost in terms of accuracy loss. Such improvements in reducing the power and area requirements are highly needed in various applications that utilizes Edge AI designs including robotic control, healthcare, and other industry applications [35-42].

## 4.2 Future works

1. Compatibility analysis. Our method only modified part of weights in convolutional layers based on the selection of rounding size, which means all other values remain the same; it has the potential to be applied after or before other methods such as parameters pruning and sparsity (DBB), BC and much more. Thus, it is worthful to find out the compatibility of our method.
2. Feasibility analysis of acceleration methods that imitate others. There are many brilliant ideas to accelerate CNNs. For example, in [21], they fused blocks of scalar PEs into a single tensor PE which turned SA to STA to achieve performance improvement. In the future, we want to find out if those methods can apply to our accelerator to improve performance further.

# BIBLIOGRAPHY


[1] Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional Neural Networks," Communications of the ACM, vol. 60, no. 6, pp. 84–90, 2017.

[2] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," 2014 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1717–1724, 2014.

[3] Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “Imagenet Large Scale Visual Recognition Challenge,” International Journal of Computer Vision, vol. 115, no. 3, pp. 211–252, 2015.

[4] K. Kapse, "An overview of current deep learned rendering technologies," 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), 2021.

[5] S. N. Chua, K. Y. Chin, S. F. Lim, and P. Jain, "Hand gesture control for human–computer interaction with Deep Learning," Journal of Electrical Engineering & Technology, vol. 17, no. 3, pp. 1961–1970, 2022.

[6] N. M. Dipu, S. A. Shohan, and K. M. Salam, "Deep learning based brain tumor detection and classification," 2021 International Conference on Intelligent Technologies (CONIT), 2021.

[7] Y. Gao and K. M. Mosalam, "Deep Transfer Learning for Image-based structural damage recognition," Computer-Aided Civil and Infrastructure Engineering, vol. 33, no. 9, pp. 748–768, 2018.

[8] M. Verhelst and B. Moons, "Embedded Deep Neural Network Processing: Algorithmic and processor techniques bring deep learning to IOT and Edge Devices," *IEEE Solid-State Circuits Magazine*, vol. 9, no. 4, pp. 55–65, Nov. 2017.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141–142, Oct. 2012.

[11] J. Yangping, "Learning Semantic Image Representations at a Large Scale", EECS Department University of California, Berkeley, UCB/EECS-2014-93, 16 May 2014.

[12] T. Wu, X. Li, D. Zhou, N. Li, and J. Shi, "Differential evolution based layer-wise weight pruning for compressing deep neural networks," Sensors, vol. 21, no. 3, p. 880, 2021.

[13] B. O. Ayinde, T. Inanc, and J. M. Zurada, "Redundant feature pruning for accelerated inference in deep neural networks," Neural Networks, vol. 118, pp. 148–158, 2019.

[14] Ma, X., Yuan, G., Lin, S., Li, Z., Sun, H., & Wang, Y. (2019). ResNet can be pruned 60×: Introducing network purification and unused path removal (P-RM) after weight pruning. 2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH). https://doi.org/10.1109/nanoarch47378.2019.181304

[15] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 3123–3131.

[16] I.-C. Lin, C.-H. Tang, C.-T. Ni, X. Hu, Y.-T. Shen, P.-Y. Chen, and Y. Xie, "A novel, efficient implementation of a local binary convolutional neural network," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 68, no. 4, pp. 1413–1417, Apr. 2021.

[17] F. Juefei-Xu, V. N. Boddeti, and M. Savvides, "Local binary convolutional neural networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 19–28, 2017.

[18] K. Ando et al., "BRein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 W," IEEE J. Solid-State Circuits, vol. 53, no. 4, pp. 983–994, Apr. 2018.

[19] H. Yonekawa and H. Nakahara, "On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA," in Proc. IEEE IPDPSW, 2017, pp. 98–105.

[20] Z. Yuan, Y. Liu, J. Yue, Y. Yang, J. Wang, X. Feng, J. Zhao, X. Li, and H. Yang, "Sticker: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-NM CMOS," IEEE Journal of Solid-State Circuits, vol. 55, no. 2, pp. 465–477, Nov. 2019.

[21] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference," IEEE Computer Architecture Letters, vol. 19, no. 1, pp. 34–37, Mar. 2020.

[22] I. Hammad and K. El-Sankary, "Impact of approximate multipliers on VGG Deep Learning Network," IEEE Access, vol. 6, pp. 60438–60444, Oct. 2018.

[23] I. Hammad, L. Li, K. El-Sankary, and W. M. Snelgrove, "CNN inference using a preprocessing Precision Controller and approximate multipliers with various precisions," IEEE Access, vol. 9, pp. 7220–7232, 2021.

[24] L. Li, I. Hammad, and K. El-Sankary, "Dual segmentation approximate multiplier," Electronics Letters, vol. 57, no. 19, pp. 718–720, 2021.

[25] I. Hammad, K. El-Sankary and J. Gu, "Deep Learning Training with Simulated Approximate Multipliers," 2019 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2019, pp. 47-51, doi: 10.1109/ROBIO49542.2019.8961780.

[26] M. Osta, A. Ibrahim, L. Seminara, H. Chible, and M. Valle, "Low power approximate multipliers for energy efficient data processing," Journal of Low Power Electronics, vol. 14, no. 1, pp. 110–117, Mar. 2018.

[27] H. Sim and J. Lee, "A new stochastic computing multiplier with application to deep convolutional Neural Networks," Proceedings of the 54th Annual Design Automation Conference 2017, Jun. 2017.

[28] "Introduction to cloud TPU | google cloud," Google. [Online]. Available: https://cloud.google.com/tpu/docs/intro-to-tpu. [Accessed: 29-Sep-2022].

[29] H. T. Kung, "Why systolic architectures?," Computer, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[30] "Adder/Subtracter," Xilinx. [Online]. Available: https://www.xilinx.com/products/intellectual-property/adder_subtracter.html. [Accessed: 23-Aug-2022].

[31] Harris, Charles R., et al. "Array programming with NumPy." Nature 585.7825 (2020): 357-362.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, K. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Advances in Neural Information Processing Systems 32, 2019.

[33] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035. Available at: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[34] Wang, Shihao, et al. "Towards current-mode analog implementation of deep neural network functions." 2022 20th IEEE Interregional NEWCAS Conference (NEWCAS). IEEE, 2022.

[35] Hammad, Issam, and Kamal El-Sankary. "Practical considerations for accuracy evaluation in sensor-based machine learning and deep learning." Sensors 19.16 (2019): 3491.

[36] Islam, Mohammad Shamim, et al. "Design of a social robot interact with artificial intelligence by versatile control systems." IEEE Sensors Journal (2021).

[37] Haidegger, Tamás, et al. "Robot-assisted minimally invasive surgery—Surgical robotics in the data age." Proceedings of the IEEE 110.7 (2022): 835-846.

[38] Hammad, Issam, Kamal El-Sankary, and Jason Gu. "A comparative study on machine learning algorithms for the control of a wall following robot." 2019 IEEE International Conference on Robotics and Biomimetics (ROBIO). IEEE, 2019.

[39] Basaklar, Toygun, et al. "Wearable devices and low-power design for smart health applications: challenges and opportunities." 2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). IEEE, 2021.

[40] Hammad, Issam, and Kamal El-Sankary. "Using machine learning for person identification through physical activities." 2020 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2020.

[41] Gültekin, Özgür, et al. "Real-time fault detection and condition monitoring for industrial autonomous transfer vehicles utilizing edge artificial intelligence." Sensors 22.9 (2022): 3208.

[42] Hammad, Issam, et al. "Using Deep Learning to Automate the Detection of Flaws in Nuclear Fuel Channel UT Scans." IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control 69.1 (2021): 323-329.

# APPENDIX

## 1. Code of LeNet-5 Model

```
#Victor Gao
#LeNet-5 model implementation
#version 3
#Modified Mar-2022
#Electrical and Computer Enginnering-Dalhousie University
from NN_lib import *
from NN_lib import _pair
from config import DEVICE,LEARNING_RATE
from myconv2d import MYconv2d

class Lenet5(nn.Module):
    def __init__(self):
        super(Lenet5,self).__init__()
        #TODO: make test on our Conv2d, thus we make copy of all the conv provide 2 method
        #to able to choose which method is gonna using.

        self.conv1=nn.Conv2d(in_channels=1,out_channels=6,kernel_size=5,stride=1)

        self.my_conv1=MYconv2d(in_channels=1,out_channels=6,kernel_size=5,stride=1)
        self.pool=nn.AvgPool2d(kernel_size=2)

        self.conv2=nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5,stride=1)

        self.my_conv2=MYconv2d(in_channels=6,out_channels=16,kernel_size=5,stride=1)
        self.pool2=nn.AvgPool2d(2)

        self.conv3=nn.Conv2d(in_channels=16,out_channels=120,kernel_size=5,stride=1)

        self.my_conv3=MYconv2d(in_channels=16,out_channels=120,kernel_size=5,stride=1)
        self.fc1=nn.Linear(in_features=120,out_features=84)
        self.fc2=nn.Linear(in_features=84,out_features=10)
        self.index=0
        self.counter_trigger=0
    def forward(self,x,TRIGGERed,count):
        #if triggered run my conv2d else run nn.conv2d
        #the nn.conv2d is to train the model and save the model
        #the my conv2d is to test the method after we got a trained model
        if TRIGGERed:
            if self.counter_trigger==0:
                print('triggered')
            self.counter_trigger+=1
            s1=time.time()
            x=self.pool(F.sigmoid(self.my_conv1(x)))
            e1=time.time()
            s2=time.time()
```

```
            x=self.pool2(F.sigmoid(self.my_conv2(x)))
            e2=time.time()
            print('time for conv1+pool is:',e1-s1)
            print('time for conv2+pool is:',e2-s2)
            x=F.sigmoid(self.my_conv3(x))
        else:
            if count:
                self.index+=1
            x=self.pool(F.sigmoid(self.conv1(x)))
            x=self.pool2(F.sigmoid(self.conv2(x)))
            x=F.sigmoid(self.conv3(x))
        x = torch.flatten(x, 1)
        x=F.sigmoid(self.fc1(x))
        x=self.fc2(x)

        return x

def run_Lenet5():
    model=Lenet5().to(DEVICE)# move model to GPU
    criterion=nn.CrossEntropyLoss()
    optimizer=torch.optim.Adam(model.parameters(),lr=LEARNING_RATE)
    return model,criterion,optimizer
```

## 2. Code of MYconv2d function

```
#Victor Gao
#LeNet-5 model implementation
#version 4
#Modified Apr-2022
#Electrical and Computer Enginnering-Dalhousie University
from NN_lib import *
from NN_lib import _pair
import glo
from torch.nn.common_types import _size_2_t
from typing import Union

def ocupied(comb_list,d,num):
    #find out if the num exist in comb[d]
  if comb_list[d]:
    for i in comb_list[d]:
      if i ==num:
        return True
    return False
  return False
def extract(comb,d):
    #extract first two number as a and b, and delete them in comb list
  a= comb[d][0]
  del comb[d][0]
  b=comb [d][0]
  del comb[d][0]
  return a,b
#Sort the orignal weight matrix, return Combinations
def sort_v2(weight,unit):#weight is a tensor
    d,c,k,j=weight.shape
```

```
    comb=[]
    old_diff=0
    for dd in range(d):
        temp_weight=weight[dd]
        temp_weight=temp_weight.reshape(c*k*j)
        sorted_,indices=torch.sort(temp_weight,stable=True)
        #next we are going to make sorted and indices as tuple
        #before that its need to convert to numpy array
        sorted_=sorted_.numpy()
        indices=indices.numpy()
        zipped=list(zip(sorted_,indices))
        temp=0
        for i in range(len(sorted_)):
            if sorted_[i]>=0:
                temp=i
                break
        neg_s=zipped[:temp]
        pos_s=zipped[temp:]
        #next step resort the negs part as descending
        neg_s=sorted(neg_s,key=lambda x:x[0],reverse=True)
        #so far we got a descending neg seq and a ascending pos seq
        n_ptr,p_ptr,p_ptr_bound=0,0,0
        cb1=[]
        cb2=[]
        while n_ptr<len(neg_s) and p_ptr<len(pos_s):#stop when n_ptr reach
the maximum
            n_val,n_loc=neg_s[n_ptr]
            p_val,p_loc=pos_s[p_ptr]
            while (n_val+p_val)<=unit and p_ptr<len(pos_s):#stop when p_val
is too big or p_ptr reach the maximum
                p_val,p_loc=pos_s[p_ptr]
                diff = abs(n_val+p_val)
                if diff<=unit:
                    if n_loc<p_loc:
                        cb1.append(n_loc)
                        cb2.append(p_loc)
                    else:
                        cb1.append(p_loc)
                        cb2.append(n_loc)
                    if diff>old_diff:
                        old_diff=diff
                    p_ptr_bound=p_ptr+1#set the new boundary
                    break #stop the current while loop, go find the next
combination
                p_ptr+=1
            n_ptr+=1
            p_ptr=p_ptr_bound
        #at this point the location for combinations are stored at n_comb
and p_comb
        cb_comb=list(zip(cb1,cb2))
        cb_comb=sorted(cb_comb,key=lambda x:x[0],reverse=False) #sort the
combinations along first element
        cb_comb=[item for sublist in cb_comb for item in sublist]
        comb.append(cb_comb)
    return comb


```

```
#Modified from Einstein Summation
def vicsum_v3(inseq_after_pad,weight):#a faster version
  n,c,h,w,k,j=inseq_after_pad.shape
  d,_,_,_=weight.shape
  out=torch.zeros(n,d,h,w)
  comb=sort_v2(weight,0.5)
  count=0

  for nn in range(n):
    for dd in range(d):
      for hh in range(h):
        for ww in range(w):

          temp_inseq_list=inseq_after_pad[nn,:,hh,ww].numpy().tolist()
          temp_inseq_list=list(chain.from_iterable(temp_inseq_list))#from
   3d->2d
          temp_inseq_list=list(chain.from_iterable(temp_inseq_list))#from
   2d->1d
          temp_weight_list=weight[dd].numpy().tolist()
          temp_weight_list=list(chain.from_iterable(temp_weight_list))#from
   3d->2d

   temp_weight_list=list(chain.from_iterable(temp_weight_list))#from 2d->1d
                templist=[]

                for i in range (0,len(comb[dd]),2):
                  pos1=comb[dd][i]
                  pos2=comb[dd][i+1]
                  out[nn,dd,hh,ww]+=(temp_inseq_list[pos1]-
   temp_inseq_list[pos2])*temp_weight_list[pos1]
                  glo.set_value(0,glo.get_value(0)+1)
                  glo.set_value(1,glo.get_value(1)+1)
                  glo.set_value(2,glo.get_value(2)+1)
                  templist.append(pos1)
                  templist.append(pos2)
                counter=0
                templist.sort()
                for ele in templist:
                  ele=ele-counter
                  del temp_inseq_list[ele]
                  del temp_weight_list[ele]
                  counter+=1
                ptr=0
                while ptr<len(temp_inseq_list):

   out[nn,dd,hh,ww]+=temp_inseq_list[ptr]*temp_weight_list[ptr]
                  ptr+=1
                  glo.set_value(0,glo.get_value(0)+1)
                  glo.set_value(2,glo.get_value(2)+1)
        return out

      def myconv2dv2(x,weight,bias,stride,pad):

        n,c_in,h_in,w_in=x.shape
        d,c_w,k,j=weight.shape
        x_pad=torch.zeros(n,c_in,h_in+2*pad[0],w_in+2*pad[0])
        if pad[0]>0:
```

```
          x_pad[:,:,pad[0]:-pad[0],pad[0]:-pad[0]]=x
        else:
          x_pad=x
        #double unfold-->window sliding based on kernel size
        x_pad=x_pad.unfold(2,k,stride[0])
        x_pad=x_pad.unfold(3,j,stride[0])
        n,c_in,h_in,w_in,k,j=x_pad.shape
        out=vicsum_v3(x_pad,weight)
        out=out+bias.view(1,-1,1,1)
        return out

      class MYconv2d(nn.modules.conv._ConvNd):
          def __init__(
              self,
              in_channels: int,
              out_channels: int,
              kernel_size: _size_2_t,
              stride: _size_2_t = 1,
              padding: Union[str, _size_2_t] = 0,
              dilation: _size_2_t = 1,
              groups: int = 1,
              bias: bool = True,
              padding_mode: str = 'zeros',  # TODO: refine this type
              device=None,
              dtype=None
          ) -> None:
              factory_kwargs = {'device': device, 'dtype': dtype}
              kernel_size_ = _pair(kernel_size)
              stride_ = _pair(stride)
              padding_ = padding if isinstance(padding, str) else
  _pair(padding)
              dilation_ = _pair(dilation)
              super(MYconv2d, self).__init__(
                  in_channels, out_channels, kernel_size_, stride_,
  padding_, dilation_,
                  False, _pair(0), groups, bias, padding_mode,
  **factory_kwargs)
          def _conv_forward(self, input, weight, bias):
              return
  myconv2dv2(input,weight,bias,self.stride,self.padding)
          def forward(self, input):
              return self._conv_forward(input,self.weight,self.bias)
```