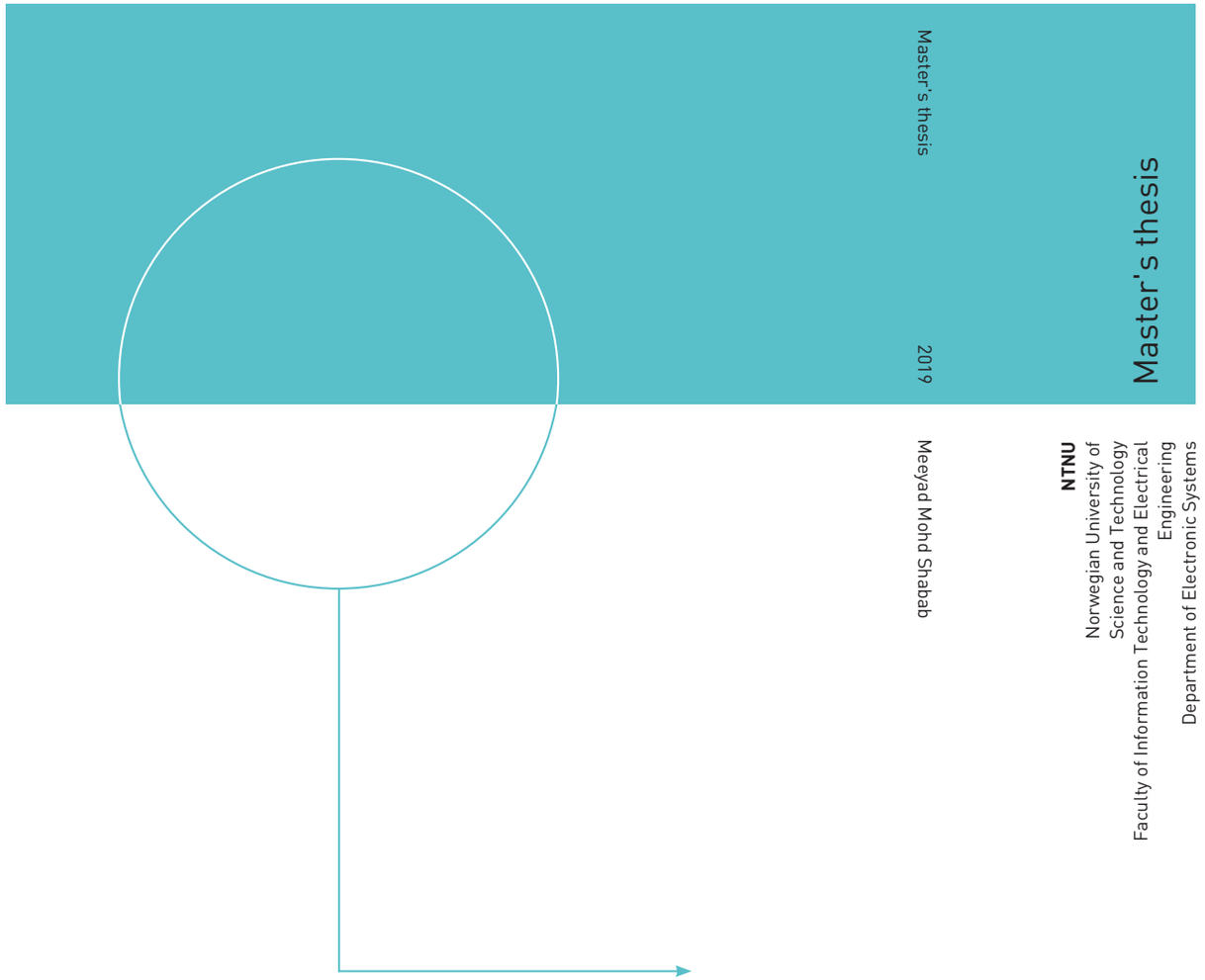


Meeyad Mohd Shabab

Artificial Intelligence in low power hardware

Analysis of Multiply-Accumulate (MAC) architecture

July 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Artificial Intelligence in low power hardware

Analysis of Multiply-Accumulate (MAC) architecture

Meeyad Mohd Shabab

Master of Science in Engineering (MSELSYS)
Submission date: July 2019
Supervisor: Prof. Kjetil Svarstad, NTNU
Co-supervisor: Omer Qadir, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronic Systems

Acknowledgements

This report is the result of the Master's thesis that is conducted during the spring of 2019. It concludes the Master of Science degree in Electronics Systems Design, with a specialization in Embedded Systems. The report is submitted to the department of Electronic Systems at Norwegian University of Science and Technology (NTNU).

The work titled "Artificial Intelligence in low power hardware" was proposed by Nordic Semiconductor as a study of implementing AI in hardware for a low power budget. Nordic also provided the necessary environment to run simulation and tests to obtain results, as well as other helpful supports. During the thesis work, knowledge was gained on AI, more specifically Convolutional Neural Networks (CNN) and its implementation in hardware. The work focuses on the Multiply Accumulate (MAC) operation of the hardware that mimics the convolution operation in CNN. This work is an extension of the project, with the same title, that was done in Autumn 2018, which studied different architectures of AI in hardware.

Special thanks goes to the supervisors Kjetil Svarstad and Omer Qadir for their valuable guidance and support in the thesis work. Also thanks to a fellow student Steinar Thune Christensen, working on similar topic, whose help has proved to be really important in the work. Also shout out to CPD department whose help is much appreciated for making the power analysis.

Abstract

This work focuses on designing an analytical model for neural network in hardware. The operation of a Convolutional Neural Network (CNN) was studied and imitated to create analytic functions in python that mimics the hardware. Analysis was done to determine the area, performance and power consumption of implementing parallel Multiply-Accumulate (MAC) units. Sparse inputs were also studied to observe their effect on MAC computations and their results are reported. The study provides the relationship between usage of MAC on area and power. The results give a general idea to the hardware designers to make expectations on the number of MAC units that can be fitted to a chip with given area and find their respective power usage.

Keywords – Artificial Intelligence (AI), Neural Network, CNN, MAC

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Objective

Machine learning (ML) and Artificial Intelligence (AI) have been gaining dramatic popularity in recent times. With sectors such as embedded software, electronics, medical sciences and so on, the application of AI increases at a faster pace. A significant part of this development is due to the use of Convolutional Neural Networks (CNNs).
Hardware manufacturers are also coming up with AI peripherals in their chips. Neural network computations are memory and power intensive. Although there exists several hardware architectures that can achieve neural network computations in less power and computation budget, but they are not as optimized. This creates a vast area in research to come up with ways in hardware solution for a more optimized neural network computation which is accurate and less power hungry.

1.2 Contribution

One of the most demanding task in a CNN is the multiplication and addition of inputs and kernels in each layer. Considering that CNNs use multiple layers of input images to improve accuracy of computation, the focus of this thesis is mostly put on this computation. In hardware, this computation is done using a Multiply Accumulate (MAC) block. The goal of the thesis is as follows:

- Create a performance model of a single MAC.
- Create a performance model of a parallel MAC based on kernel size.

1.3 Methodology

This work involves writing code blocks in python that mimics the hardware scenario. This was done to create an analytical model that supports the computation of convolution operation on a hardware MAC unit. The created model in Python was used to study the patterns of area and power consumption for implementing parallel MAC units, considering both normal and sparse inputs. Similar operation was done, writing codes in Python, to generate random 1-D and 2-D matrix for inputs and perform computation that is followed in CNN to get an output feature map within each layer. The output is compared to the output of the python model that was created to perform the work in thesis which matched as expected.

1.4 Report Structure

Chapter 1 - Introduction

Chapter 1 introduces the work - its motivation, the target problem for which the work was necessary and the contributions made in this thesis.

Chapter 2 - Background

Chapter 2 gives a brief overview of CNN and it operation that is relevant to this study.

Chapter 3 - Related Work

Chapter 3 presents the related work done in this field that are an in relation or extension to the work done in the thesis.

Chapter 4 - Architecture

Chapter 4 presents the architecture that was designed and implemented for the study to get a result for analysis.

Chapter 5 - Analysis

Chapter 5 compares the result that is obtained from running the codes and discusses about the data and their relevance.

Chapter 6 - Discussion

Chapter 6 discusses about the work that is done and also highlights the future scope that can be implemented to further extend the study.

Chapter 7 - Conclusion

Chapter 7 mentions the conclusion of the work along with some lessons learned and suggestions that can be used as a starting point if the study is chosen to be continued further.

Abstract

Contains the source codes and parameters used to conduct the study. Graphs from analysis are also included in this section for better visualization.

2 Background

In this chapter some background and theory necessary to understand the subject is presented. At first, machine learning and neural networks are introduced, including some theory on training the network. Then a brief explanation of CNN is presented along with the basics behind multiply accumulate operation, which is the prime focus of this thesis work.

2.1 Machine Learning

Machine learning (ML) stems from computer science and was originally defined in 1959 by A. L. Samuel [8] as *a field of study that gives computers the ability to learn without being explicitly programmed.* A more formal definition was made by Mitchell [9] as :

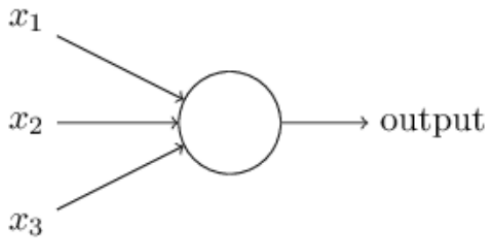
> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at task in T, as measured by P, improves with experience E.*

One such example is the hand writing recognition program. The task T will be to recognize and classify the handwritten words within images, the performance measure P is the percentage of words correctly classified, and training experience E is the database of words with known classification. There are many applications of machine learning such as image recognition, object detection, speech recognition etc. and the list continues to grow.
A subset of ML is neural network, also known as Deep Neural Network (DNN) when the network consists of multiple layers. A further extension of DNN is the Convolutional Neural Network (CNN) which is the highlight of this thesis.

2.2 Neural Network

A neural network is a network or circuit of neurons and is composed of artificial neurons or nodes. Neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network for solving AI problems. The term neural network

has its origin in attempts to find mathematical representations of information processing in biological systems, where the perceptron was one of the big influential outcomes of this research [10]. The perceptron is an artificial neuron that was developed by Frank Rosenblatt during the 1950s and 1960s. It takes in several binary inputs, x_1, x_2, x_3, ..., and produces a single binary output. This is shown in figure 2.1 along with the respective mathematical representation.



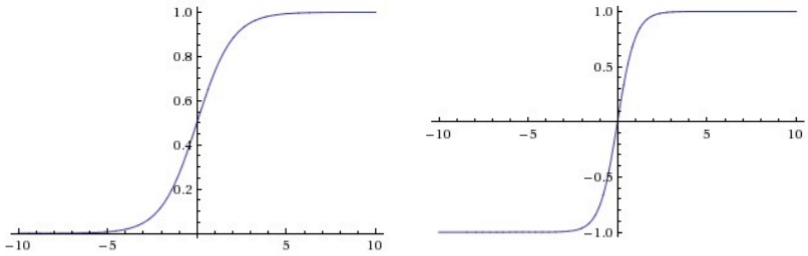
$$output = \begin{cases} 0 \; if \; \sum_j w_j x_j \leq threshold \\ 1 \; if \; \sum_j w_j x_j > threshold \end{cases}$$

Figure 2.1: Perceptron model and equation

Rosenblatt introduced weights and threshold value, which are all real numbers and parameters of the perceptron. Based on these parameters the output will either be 0 or 1, depending on the input as demonstrated in the figure above.
Current models are similar to Rosenblatt's perceptron, with a few changes. Bias, b is introduced instead of using threshold and is defined as $b \equiv -threshold$. Additionally, an activation fucntion is introduced which allows small changes in the weights or bias to only cause a small change on the output, this property is helpful when training a network. For the perceptron such small changes may cause the output to flip, e.g. from 0 to 1. The new model and definition of a neuron is demonstrated in the following equation:

$$y = f(w \cdot x + b) = f(\sum_j w_j x_j + b) \tag{2.1}$$

In equation 2.1, y is the output that is given by the dot product of neuron's weight vector w, and the input vector w, plus the bias. The function, *f()*, that is wrapped around is known as the nonlinear activation function. The three most common activation functions are **sigmoid**, **tanh** and **RelU** and are shown in figure 2.2. Activation function is not a major concern in this work so it will not be discussed in details.

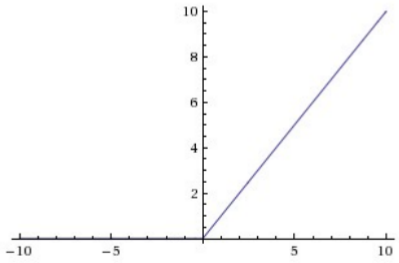


$$\sigma(z) = \frac{1}{1+e^{-z}}$$

(a) Sigmoid

$$tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

(b) tanh



$$RelU(z) = max(0, z)$$

(c) RelU

Figure 2.2: Activation functions

A neural network consists of many layers that are organized in multiple layers. The first layer of the network in the input layer, that is followed by one or more hidden layers. The term hidden layers are used as these layers are neither inputs nor outputs. The last hidden layer is followed by the output layer. The number of neurons in the output layer depends on the task. For example. in handwritten number recognition, there would be 10 output neurons; one for each number ranging from 0 to 9.

If all the inputs to a layer's neurons stems from the previous layer, it is known as a *feed-forward* neural network. If connections between neurons can form a directed cycle in the network, it forms a *recurrent* neural network. Further discussion on the network types are not important for this work, hence they are not touched in details.

2.3 Network Training

Training a network involves a way to optimize the weights and biases of the network. This work doesn't focus on training the network so network training will be discussed in brief. For training a network, a set of input vectors x_n, where *n 0 1, 2, ..., N*, together with a corresponding set of target vectors x_n, is required. A cost function, *C(w,b)*, is introduced and to get better classification of results, the cost function needs to be minimized. The equation for cost function is as follows:

$$C(w,b) = \frac{1}{2N} \sum_{n=1}^{N} ||y(x_n) - t_n||^2 \tag{2.2}$$

In equation 2.2, **w** and **b** is the collection of all the weights and biases in the network. To minimize the cost function, an algorithm called *gradient decent* is used. The idea behind it is to alter the values of the weights and biases by updating them with small steps in the direction of the negative gradient. This update for each weight component w_k and p_l is given by:

$$w_k \rightarrow w_k = w_k - \eta \frac{\partial C(w,b)}{\partial w_k} \tag{2.3}$$

$$b_l \rightarrow b_l = b_l - \eta \frac{\partial C(w,b)}{\partial b_l} \tag{2.4}$$

These updates are performed many times for the cost function to converge towards a local minimum. The parameter η is the learning rate that decides how fast the cost function converges. Larger η value causes the cost function to increase and not converge. η is known as the hyper-parameter and is not trained like the weights and biases, but can still be chosen appropriately and possibly be fine tuned [11].

Duration of training time depends on the number of training inputs. *Stochastic gradient descent* can be used to speed up the process, where instead of using all N training inputs, a smaller number of samples are randomly chosen from the training set. This makes the update operation from equations in 2.3 and 2.4 perform faster.
Back propagation is another most important algorithm to train neural network in recent times as it provides faster way of computing the gradient of the cost function. First an

input vector x_n is passed forward in order to find the activation of all the neurons. Then error values on the outputs are propagated backwards through the network, that is used to calculate the gradients and perform the updates [12][10].

2.4 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a feed-forward neural network, which means that there is no loop-backs in the network like in back propagation. CNNs are very similar to normal neural networks as they are made up of neurons which have trainable weights and biases. The difference is the CNN has fully connected layer with additional two layers: *pooling later* and *convolutional layer*. These layers are stacked several times to form a CNN [11]. This work explicitly focuses on the *convolutional layer* and uses its operation to be implemented in a hardware environment.

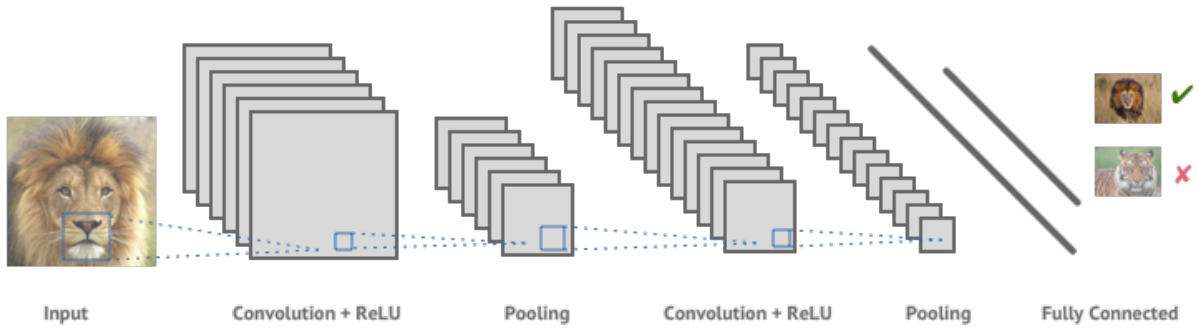


Figure 2.3: Convolution Operation[1]

Figure 2.3 shows an example of a CNN that consists of 6 layers (excluding the input). CNN uses filters (also known as kernels) to detect features such as edges present throughout an image. A filter is a matrix of values, known as weights, that are trained to detect specific features. The filter moves over each part of the image to check if the feature which it is supposed to detect is present. To provide a value representing the confidence about the presence of the specific feature, the filter performs a convolution operation, which is an element-wise product and sum between two matrices.
If the feature is present in the part of the image, the convolution operation between the filter and that part of the image results in a real number with a high value. If the feature is not present, the resulting value is low.

Additionally, a filter can be slid over the input image at varying intervals, as represented in Figure 2.4, using a stride value. The value dictates by how much the filter should move each step. Sometimes padding is added, as shown in Figure 2.4, for the filter to capture all the data within the input image. The filter is passed through a non-linear mapping so that the CNN can learn the values for a filter that detect features present in the input data. The output of the convolution operation is summed with a bias term and passed through a non-linear activation function. This introduces non-linearity in the network and is done by the rectified linear unit (ReLU), which turns the values that are less than zero to zero and all the positives are left unchanged. The final stage of CNN is the pooling layer that is down-sampling of a feature map. Pooling layer operates on each feature map independently.The most common approach of pooing is max pooling.

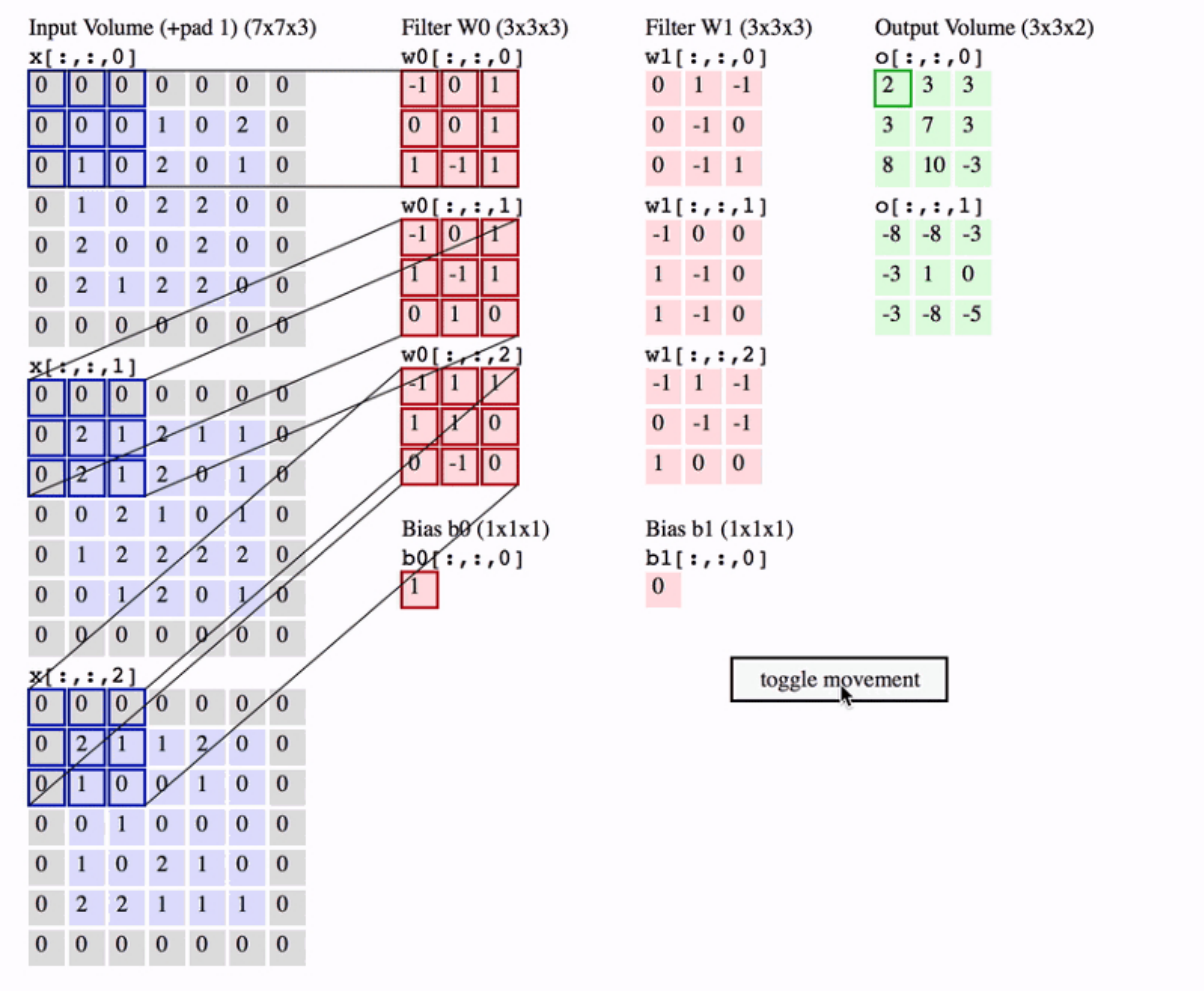


Figure 2.4: A demo of a Conv layer with K = 2 filters, each with a spatial extent F = 3 , moving at a stride S = 2, and input padding P = 1. [2]

2.5 Multiply Accumulate (MAC)

Multiply accumulate operation is one of the vastly used operation in computing, especially in digital signal processing. It is used to compute the product of two numbers and adds that product to an accumulator which is usually a memory. The hardware unit that does the operation is known as multiplier-accumulator (MAC or MAC unit [13]). The operation can be represented by equation 2.5.

$$a \leftarrow a + (b \times c) \tag{2.5}$$

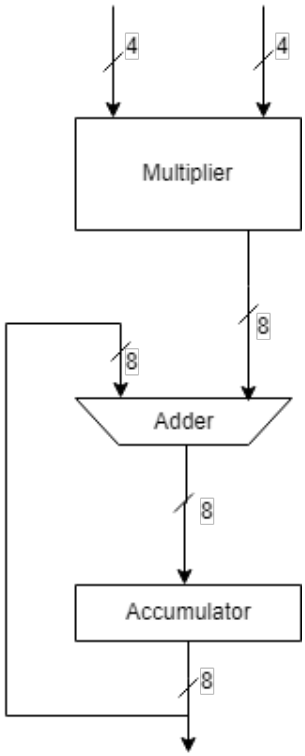


Figure 2.5: 4-bit MAC block diagram

Figure 2.5 represents a MAC unit that takes in two inputs (b, c from equation 2.5) of 4-bits each to the multiplier. The output of the multiplier is 8-bit due to the product of two 4-bit inputs in the multiplier. The output of the multiplier is fed to the adder circuit which adds the value of the multiplier from the previous cycle and stores it to the accumulator. The adder circuit is further split-up to represent its architecture in Figure 2.6.

Figure 2.6 (a) represents a 4-bit adder circuit block diagram which is composed of 4 full

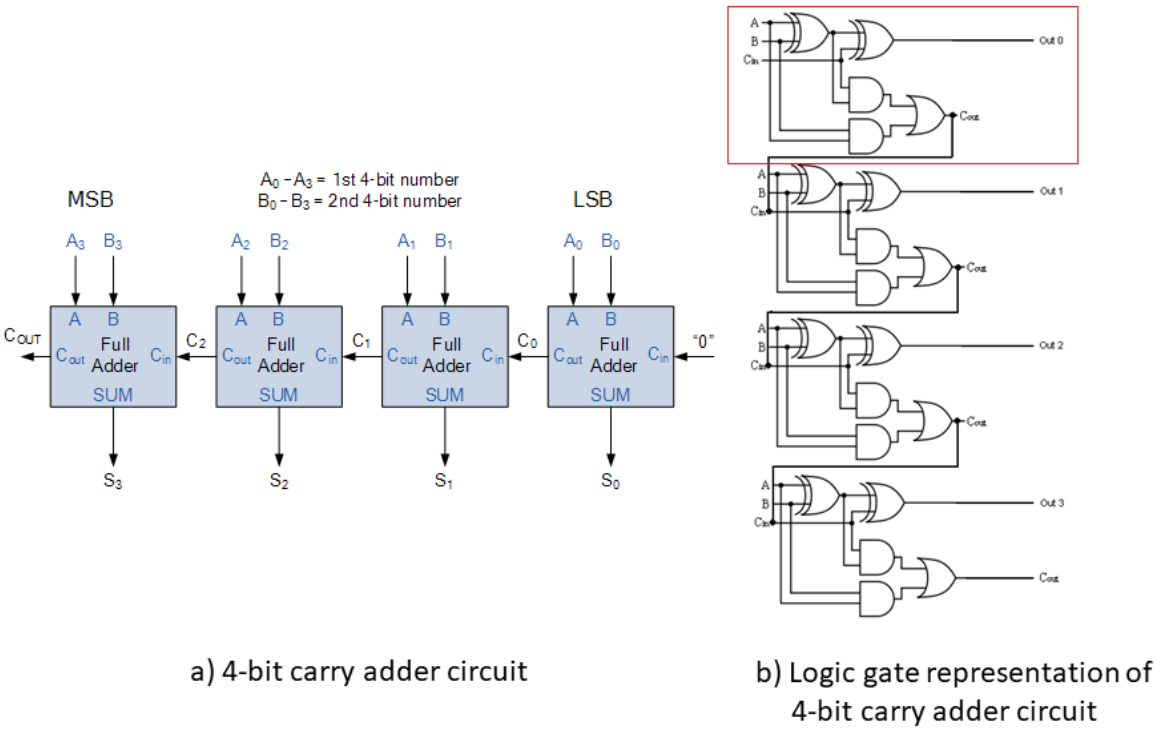


Figure 2.6: 4-bit Adder Circuit [3, 4]

adder circuits and each of these adder consists 2-XOR gates, 2-AND gates and 1-OR gate as shown in Figure 2.6 (b) (squared portion). Each adder outputs a summation value of 2 1-bit input (A_0, B_0, ...) and a carry value that is propagated to each of the connected full adders until the end.

This project concentrates on the logical design of each of the blocks in the MAC unit for which a low level representation of the blocks are discussed in the background section.

2.6 Sparse Matrix

In mathematics "sparse" and "dense" often refers to the number of zero vs. non-zero elements in an array (vector or matrix). A sparse array is one which contains mostly zeros and few non-zero entries, while a dense array contains mostly non-zero values.
In the context of neural networks things that are described as sparse or dense include the activation of units within a particular layer, the weights and the data. In sparse connectivity, a small subset of units are connected to each other with no connection with units that has zero weights.

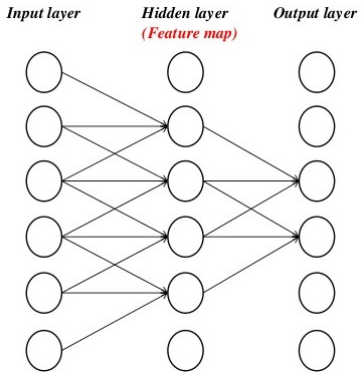


Figure 2.7: CNN Sparse Connectivity Representation [5]

Sparse Connections, as presented in Figure 2.7, is one of the focus of this study as it saves power. Since sparse connections does not deal with the zero input values, they are omitted. This means that the multiply-add operation will not be performed for zero values, saving number of cycles, gate usage and thus saving power as the MAC block would be turned off for zero inputs. More details of the operation would be discussed in the *Methodology* chapter.

3 Related Work

A lot of research had been conducted on making the MAC operation more efficient with designs and methods that make the operation faster. This section focuses on a couple of those works that has been done on the CNN MAC designs, so that it can be used as a base to support the work done in this thesis. Also the works in this chapter can be considered as a motivation for future work.

3.1 Low complexity MAC

3.1.1 Abstract

CNNs require large amounts of processing capacity and memory bandwidth. Typical hardware accelerators have large numbers of MAC units. Multipliers are large in integrated circuits (IC) gate count and power consumption. "Weight sharing" accelerators have been proposed where full range of trained weights are compressed and put into bins and it's index is used to access the weights-shared value. Parallel accumulate shared MAC (PASM) [6] is implemented is discussed in this paper, that is coupled with the weight-shared CNN method. PASM re-architects the MAC to count the frequency of each weight and place it in a bin. The accumulated value is computed in a subsequent multiply phase that significantly reduces gate count and power consumption of the CNN.

3.1.2 Concept

The PASM architecture reduces oiwer and area by first making the MAC do the accumulation first, followed by a shared post-pass multiplication [6]. The accelerator is shown in Figure 3.1. PASM has two phases:

1. Accumulate the image values into weight bins (known as parallel accumulate and store (PAS)).
2. Multiply the binned values with the weights (completing the PASM).

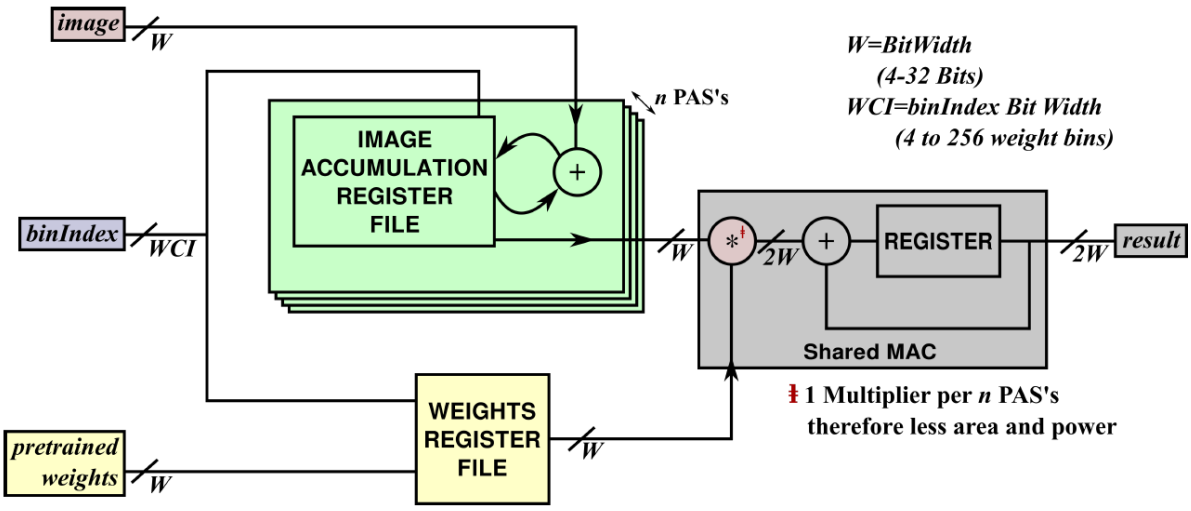
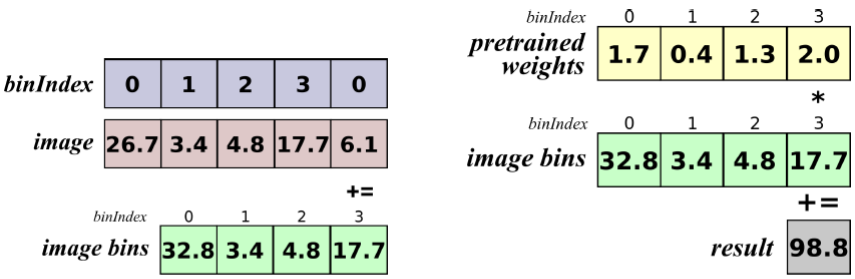


Figure 3.1: PASM showing PAS unit followed by a shared MAC[6]

Figure 3.2(a) represents an example of the accumulate phase. The PAS unit is a sequential circuit that consumes a pair of inputs (the *image* value and the *binIndex* value) at every cycle. The binIndex is the index of the weight value in the dictionary of weight encoding. The PAS unit contains B accumulators, one for each entry in the dictionary of weight encoding. The accumulator is initially set to zero. Every time the PAS consumes an input pair, it adds the image value to the accumulator with the index in binIndex.

Figure 3.2(b) shows the example of the second phase of PASM operation. Here the histogram of weight indices is combined with the actual weight values to compute the result of the sequence of multiply-accumulate operation. Pre-trained weight of one bin is multiplied with the image value of the same bin (e.g. 1.7×32.8 of bin 0 for both weight and image gives 55.76). Subsequent bins of weights and images undergo the same operation until all the corresponding bins are multiplied and accumulated in the *result* register. The second phase of PASM can be implemented using a traditional MAC unit.



(a) Phase 1: As each image value is streamed in, its associated bin index is also streamed so that the image values can be accumulated into correct bins.

(b) Phase 2: Each bin accumulated value is multiplied with its corresponding pre-trained weight value to produce the final result.

Figure 3.2: PASM Operation[6]

3.2 Vedic MAC

3.2.1 Abstract

In this paper[7] a MAC design is presented using vedic multiplier with square root carry select adder (SQRT-CSLA). It was seen to have significant impact on the area and power consumption also providing better performance of the entire neural network.

3.2.2 Concept

3.2.2.1 Vedic Multiplier

Speed and accuracy is the constraint in multiplication process[14]. Speed can be achieved by reducing the computation process in the multiplication technique which can be efficiently done by a *vedic multiplier* [15]. The architecture of a 8×8 vedic multiplier is presented in Figure 3.3.

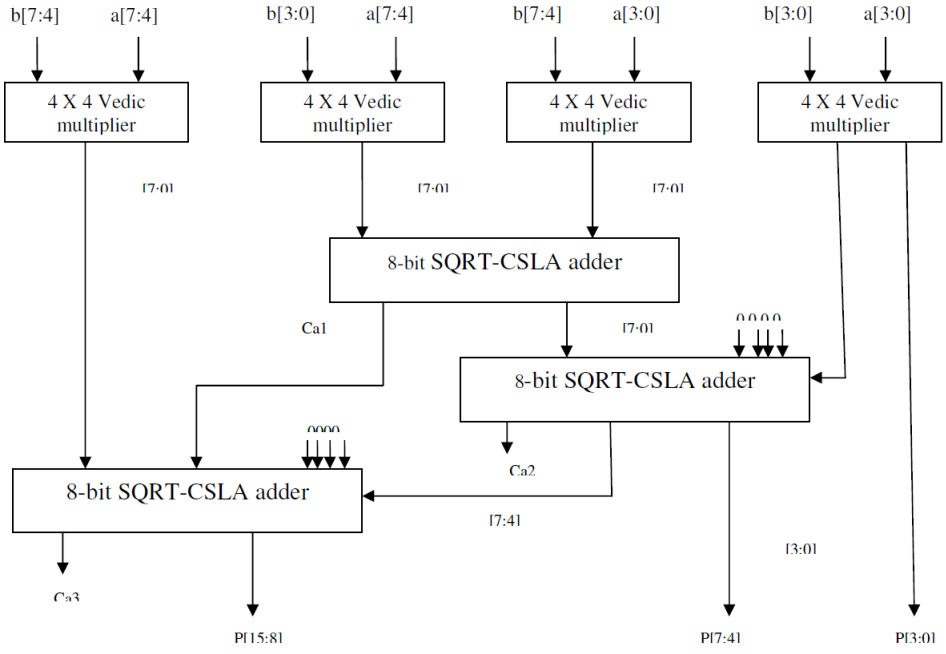


Figure 3.3: Block Diagram of 8 x 8 Vedic Multiplier[7]

In Figure 3.3, each 4×4 vedic multiplier perform the operation separately, computing partial products that are added by the 8-bit SQRT-CSLA; finally giving out a 16-bit

multiplication output. The 8-bit input sequence is divided into two 4-bit numbers and a combination of the inputs are fed in 4 4 × 4 vedic multiplier to perform efficient Vedic multiplication[16]. The inputs of the 4-bit multipliers are a[7:4] b[7:4], a[3:0] b[3:0], a[7:4] b[3:0] and a[3:0] b[7:4].

3.2.2.2 SQRT-CSLA Adder

Carry propagation delay and low complexity are recognized as high potential in every addition circuit[14]. An efficient output can be achieved by the SQRT-CSLA adder architecture. Based on the selection of carry inputs, there are two kinds of SQRT-CSLA adder: a) Dual Ripple Carry Adder (RCA) based SQRT-CSLA; d) Binary to Excess-1 converter (BEC) based SQRT-CSLA.

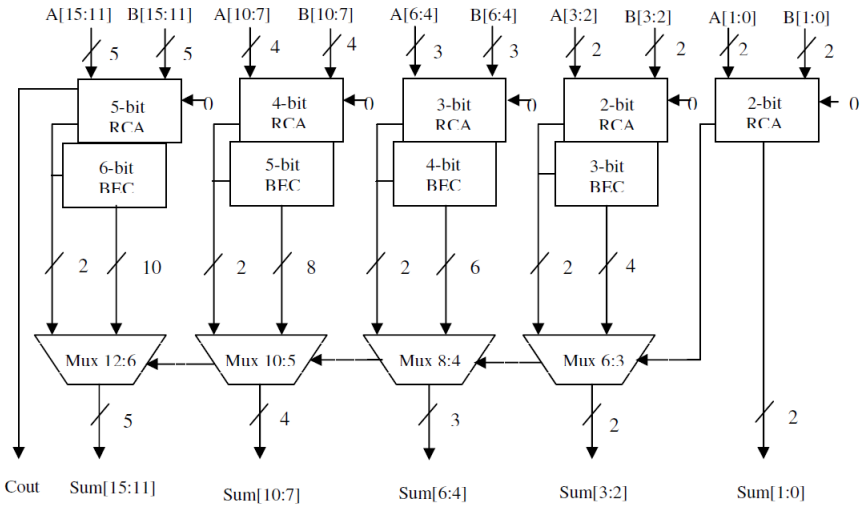


Figure 3.4: Architecture of BEC based SQRT-CSLA[7]

Figure 3.4 represents the architecture of a BEC based SQRT-CSLA containing BEC, RCA and mux. Half adders, full adders and multiplexers are used for providing partial product addition results. BEC circuits are used to provide same RCA functions, but have different architectures with less gate count. Due to increasing propagation delay in the RCA circuit, BEC based SQRT-CSLA adder architecture is used in the design of a vedic MAC unit.

4 Architecture

This chapter discusses about the design choices made to conduct the thesis work, along with the explanation of why such choices were made. The reasons behind the design choices have been explained and discussed with reasoning. The study is done using python programming and keeping the hardware design in mind so that the approximation of the design can follow the real world hardware implementation.

4.1 Design choices

4.1.1 Multiply Accumulate (MAC) Design

The prime focus of this project is to design an analytical model for the computation of convolutions performed in CNN. This is typically multiplication of two inputs (image value and weight value) and addition of these products with its previous values. The multiply-add is performed in CNN can be implemented using a MAC unit in hardware. Traditional MAC unit is used in this study due to the complexity of AI networks and limited time-frame. The re-architectured MAC units and methods mentioned in **Chapter 3** can be used as a motivation to create a MAC block that is specialized for neural network computation. It is also easy to measure the required objectives of this study using a traditional MAC unit with the given limited time.

4.1.2 Programming Language

The main object of this thesis is to design a hardware MAC architecture with minimal area and power consumption. For hardware design hardware description language (HDL) is the go to language. But due to the complexity of the hardware design for neural network and time constraint, Python is used to design methods and code blocks that mimic the actual hardware scenario. Also python has excellent support and is the mostly preferred choice of language for creating AI model and CNN.

4.1.3 Design Parameters

Since the work involves creating code blocks using python, certain parameters are used to demonstrate performance, area and power calculations. All these values are taken from the libraries that Nordic uses for its 55nm technology. No simulation software is used which makes it hard to get exact power consumption of the functional units.
For Area calculation, the total number of logic gates are found and their respective sizes are computed for different MAC designs. To calculate the performance, the number of cycles that is required by the MACs to complete the whole operation is considered. Power, more specifically dynamic power, is calculated using the formula $\frac{1}{2} \times \alpha \times C \times V^2$. Dynamic power is one of the major contributor of power consumption for transistors and it increases linearly with the area, which is why dynamic power is considered.

4.2 Design explanation

4.2.1 Multiply-Accumulate (MAC) Unit

The MAC unit consists of a multiplier, an adder and an accumulator which is usually a memory or register. A block diagram of MAC unit is presented in Figure 2.5. A typical 4×4 multiplier unit is presented in Figure 4.3 which takes in two 4-bit inputs and produces an 8-bit product. The multiplier unit is composed of 2 4-bit full-adder blocks whose logic gate representation is shown in Figure 4.2. Figure 4.3 presents the complete architecture of a 4×4 MAC unit. It is constructed with a 4×4 multiplier unit, an 8-bit adder unit and an accumulator unit (typically a memory unit) that holds the output value of the adder unit and loops it back to the adder unit. This is done so that the current value of the adder unit at time t is added to the previous value of the adder at time *t-1*. The carry out from the 8-bit adder can be used to check if the data has been reached out of frame.

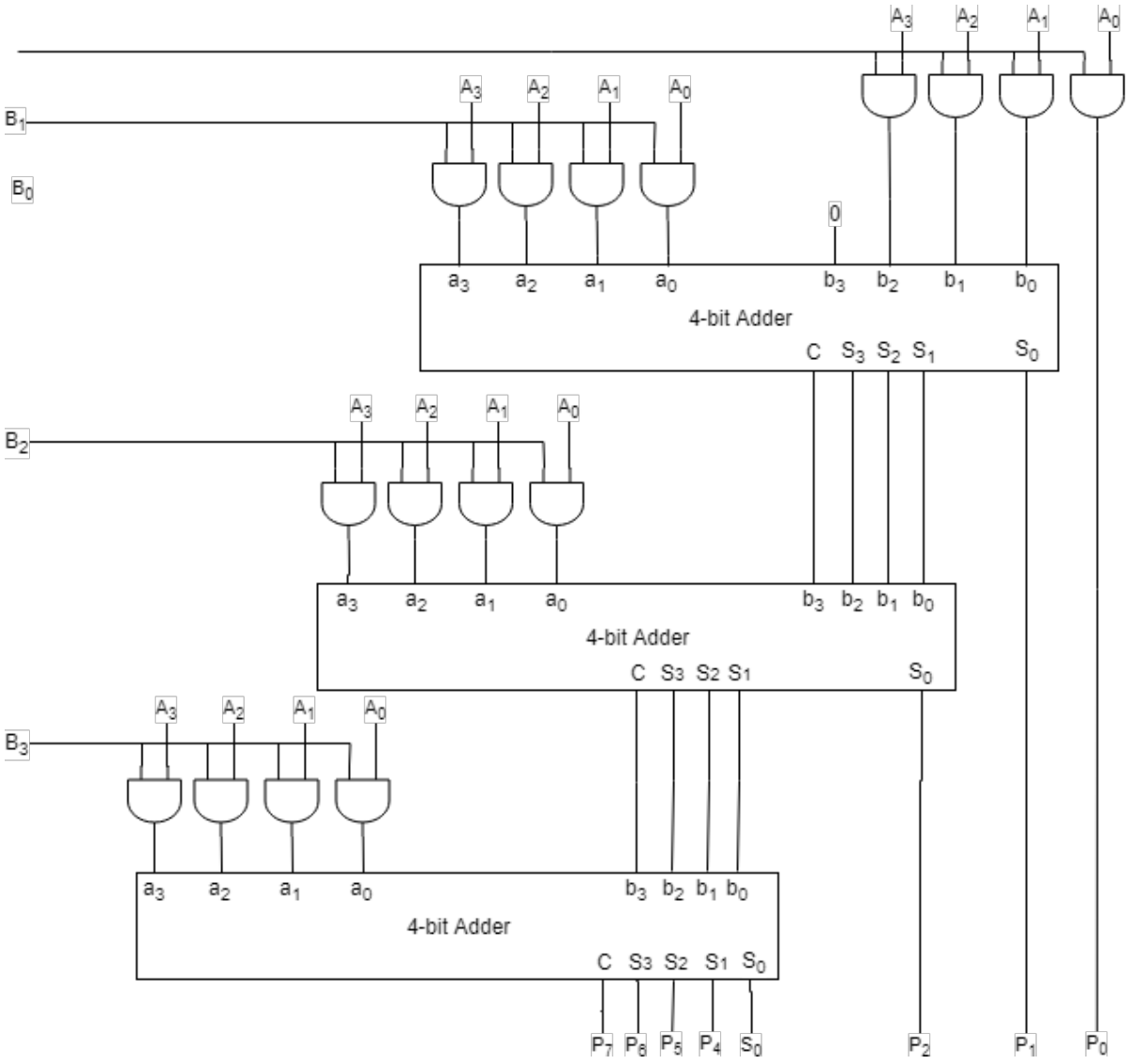


Figure 4.1: Architecture of a 4-bit Multiplier unit

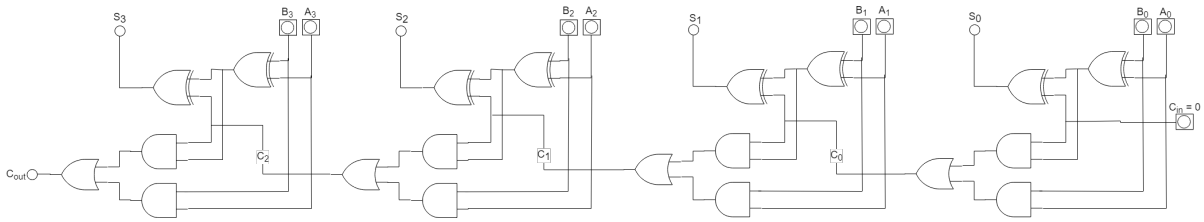


Figure 4.2: Logic gate representation of a 4-bit Full Adder

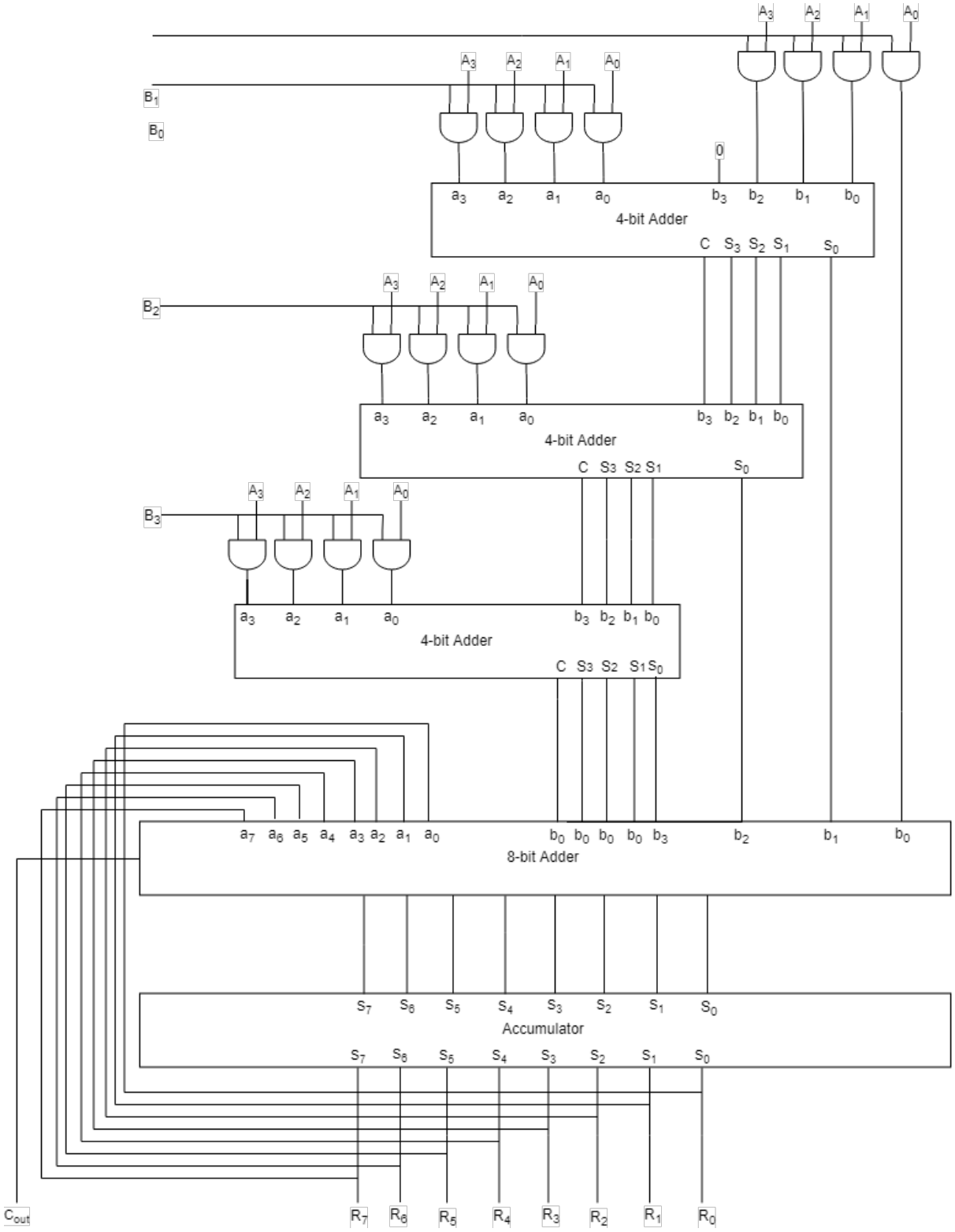


Figure 4.3: Architecture of a 4-bit MAC unit

The 4-bit MAC block is discussed to give an overview of how a MAC is constructed. The size of the MAC block would increase in size based on the number of inputs. Since the calculation of *Area* is based on the number of gates in the circuit, table 4.1 represents

the number of gates based on the n-bit input of MAC, where n = 1,2,3...N. For CNN it is seen that maximum word length of 16-bit accommodates almost all forms of data representation, so more than 16-bit word length is not explored in this study.

	4-Bit MAC	8-Bit MAC	16-Bit MAC
AND	56	208	800
XOR	40	144	544
OR	20	72	272

Table 4.1: Table representing number of logic gates based on word length

4.2.1.1 Area

To calculate the Area of the MAC unit, the traditional hardware MAC design was inspected to get the number of logic gates. Figure 4.1 through 4.3 shows the logical components of a 4-bit MAC unit. It has 16 AND gates and 3 4-bit Adder units that makes up the multiplier and a 8-bit adder unit that deals with the summation. The number of adder units in the multiplier is *n-1* where n is the number of inputs in the MAC unit. Each 4-bit adder unit is composed of 8 AND gates, 8 XOR gates and 4 OR gates. Following the same construction, the 8-bit adder unit has twice the same number of logic gates. All these values adds up to 56 AND gates, 40 XOR gates and 20 OR gates which makes up a 4-bit MAC unit. Similar rules are followed for calculating the total number of logic gates in a 8-bit and 16-bit MAC and is presented in Table 4.1.

Values of area occupied by each 2-input logic gate was collected from Nordic's 55m library and is presented in Appendix A1. These values were considered to calculate the area of a MAC unit. The area of MAC is calculated for both normal input data and sparse input data. The sparse input data is generated based on the threshold value 2^n bits. If the word is 8-bit long then the threshold value is set to $2^8 = 256$, beyond which all the values are turned to zero using python's numpy library. The difference in area of a traditional MAC block and that of a sparse MAC block is discussed in the **Analysis** chapter.

4.2.1.2 Performance

The performance of the MAC blocks is calculated based on the number of cycles of MAC code blocks that is run in the program. At first, the number of cycles taken for a single MAC block is computed. Considering the number of MAC that are implemented, the number of cycles are calculated. For instance if a single MAC block takes 324 cycles to calculate a 8×8 input image with a 3×3 kernel, then using two MAC blocks will take half (162 cycles) the amount of cycle to perform the same same operation doubling the performance.
For parallel functionality, functions are written that uses a maximum of k^2 MACs for a $k \times k$ kernel; i.e. a 3×3 kernel will have a maximum of $3^2 = 9$ MACs to calculate a value of the output feature map at one cycle, which would take 9 cycles if a single MAC is used. A detailed discussion of this is done in **Chapter 5**.

4.2.1.3 Power

- Calculating power using gate
- Calculating power using Area

4.2.2 Program Structure

This section explains the functionality of each of the code blocks that were written as part of work in the thesis. The code blocks are written keeping the hardware implementation is mind. The values obtained from the codes are an approximation to the real implementation. The codes can be used as a reference point for design in HDL.

4.2.2.1 Initial Block

Figure 4.4 shows the block of python code that is used to initialized all parameters used in the program. Random matrix for both image and kernel are created using numpy library in python. The dimension of both matrices are controlled using the row and column variable *(dimIR, dimIW)* and *(dimWR and dimWC)* respectively. The size of the image

```
#---------------- Library imports ---------------------------
import numpy as np
import math
from numpy import array, reshape, count_nonzero
from collections import namedtuple
#----------------- Variable declaration ----------------------
dimIR = 8 #row-dimension of input image matrix
dimIC = 8 #column-dimension of input image matrix
dimWR = 3 #row-dimension of kernel/weight matrix
dimWC = 3 #column-dimension of input kernel/weight matrix
cnt = 0 # for counting the number of MAC operations
cnt_sp = 0 # for counting the number of sparsed MAC operations
cntCycle = 0 # for getting the number of cycles within one MAC operation | initialized to zero whenever MAC function is called
cycle = 0 # for geting the number of MAC operations based on strides without cosidering sparsed input data
cycle_SM = 0 # for geting the number of MAC operations based on strides cosidering sparsed input data
bitSwap = 8 # variable to control numbber of gates | 8 for 8-bit representation and 6 for 16-bit representation

#------- Creating the input matrix ---------------------------
iM = np.random.randint(0, 500, size = (dimIR, dimIC)) # random generation of input image matrix
kM = np.random.randint(-1, 5, size = (dimWR, dimWC)) # random generation of kernel matrix

# printing for debugging purposes ---
print(iM)
#print(kM)
#-------------------------------------

# calculating the number of stride for CNN operation based on the dimension of image and kernel
strides = (dimIC - dimWC) + 1

# logic to handle stride value based on kernel dimention to fit 1-D/2-D array
if (dimWC != dimWR):
    stride = strides + 1
else:
    stride = strides
```

Figure 4.4: Snippet of Initialization code block

and kernel matrix are manipulated changing these values to represent both 1D and 2D array. The stride of the kernel on image is controlled by the *stride* variable. The strides are controlled using if-else control statement for the code to work for 1-D and 2-D array.

```
# Mac struct to define the design of MAC using logic gates
MacStruct = namedtuple("MacStruct", "OR XOR AND")

# variable declaring total number of gates for one n-bit MAC
gate = MacStruct(totalORGates, totalXORGates, totalANDGates)
```

Figure 4.5: Code snippet of MAC gate structure

Figure 4.5 represents the code snippet that defines the gate structure in a MAC unit. *MacStruct* is a tuple that defines a struct known as *MacStruct* that consists of AND, OR and XOR gates. This structure is used to obtain the number of logic gates generated by running the MAC function which is used to calculate area and power consumption by the MAC unit.

The function in Figure 4.6 represents the calculation of area for the MAC unit. It takes in one parameter **MACs** which is the number of MACs used. The maximum number of MACs used is the size of the kernel dimension. For instance, if a 3×3 kernel is used

```
# Function for Calculating Area
def MAC_Area(MACs):
    Area = (totalANDGates * And + totalORGates * Or + totalXORGates * Xor) * MACs
    return Area
```

Figure 4.6: Snippet of code block for calculating Area

then the maximum MACs used for parallel operation is 9. The global variables *And, Or, Xor* contains the respective area values for 2-input logic gates as collected from 55nm library. The area of the MAC unit is then calculated by adding the product of each logic gate instances with it's respective area of the logic gate and finally multiplying with the number of MAC instances giving an approximate value in μm^2.

```
# Function for calculating power considering area
def Power(Area):
    global voltage, frequency, activity_factor, capacitance
    power = 0.5 * activity_factor * capacitance * (voltage * voltage) * Area
    return int(power)
```

Figure 4.7: Snippet of code block for calculating Power

Figure 4.7 shows the code snippet for calculating dynamic or active power. For power calculation, values of voltage, activity factor and average routing length is considered as per 55nm technology in the Nordic components library. These values are mentioned in the Appendix. Using all the parameters, the power is calculated considering the frequency of 100MHz.

```
# Parallel MAC functional block conducting MAC operation of the same size as kernel dimension
def MAC(a,b):
    global acc, cnt, cycle, cntCycle
    global And, Or, Xor

    for item in range(len(a)):
        mul = a[item]*b[item]
        cntCycle += 1
        acc = acc + mul
        cntCycle += 1
        cnt +=1 #getting the mac operation considering each value
    cycle += 1 #getting the number of cycles for whole operation based on kernel size

    OrGate = gate.OR * cnt
    XorGate = gate.XOR * cnt
    AndGate = gate.AND * cnt

    return (acc, cycle, OrGate, XorGate, AndGate, cntCycle, cnt)
```

Figure 4.8: Code snippet of Parallel MAC function

The implementation of parallel MAC operation is presented in Figure 4.8. This function

performs the multiply and addition operation that is done by a traditional hardware MAC unit. The function also performs parallel operation to improve performance as opposed to using a single MAC block.

```
# Parallel MAC operation fucntion considering sparsed input
def MAC_SP(im_in,k_in):
    global acc, cnt_sp, cycle_SM, cntCycle
    global And, Or, Xor

    for item in range(len(im_in)):
        if (im_in[item] == 0 and k_in[item]):
            pass
        else:
            mul = im_in[item]*k_in[item]
            cntCycle += 1
            acc = acc + mul
            cntCycle += 1
            cnt_sp += 1
        cycle_SM += 1

    OrGate = gate.OR * cnt_sp
    XorGate = gate.XOR * cnt_sp
    AndGate = gate.AND * cnt_sp

    return (acc, cycle_SM, OrGate, XorGate, AndGate, cntCycle, cnt_sp)
```

Figure 4.9: Code snippet of Parallel MAC function consider sparse input

Figure 4.9 demonstrates parallel MAC operation considering sparse input. The difference between this block and a traditional MAC unit is that, a sparse MAC unit does not consider any zero input values. It skips over those values which reduces the number of MAC operations and thus reduce the power consumption altogether.The sparse representation in this thesis is done to show the reduction of MAC operation.

```
iM[iM > 256] = 0
cnt = 0
print(iM)
# calculate sparsity
sparsity = 1.0 - (count_nonzero(iM) / iM.size)
print("Sparsity of input data = %s" %sparsity)
```

Figure 4.10: Code snippet for sparse input formation

Formation of sparse input is presented in Figure 4.10. Based on the word length, the threshold value is put in the *iM[iM > 256]*. The code snippet shows the threshold value set to 256 ($2^8 = 256$) considering a 8-bit word length, or 8-input MAC design. The sparsity of input is calculated to compare the MAC performance by changing the word length and input size.

4.2.2.2 Operational Blocks

```
# operation block performing convolution operation using a single MAC
cnt_single_mac = 0
outFS_v = []
mac_cycle = 0

for strideY in range(stride):
    for strideX in range(strides):
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                #print(IM, K)
                mul = IM * K
                acc = acc + mul
                cnt_single_mac += 1
            mac_cycle += 1
        outFS_v.append(acc)
outFS = array(outFS_v)
outFS = outFS.reshape((int(outFS.shape[0]/strides), strides))
#print(outFS)
MAC_Single_Area = MAC_Area(1)
Power_Consumed = Power(MAC_Single_Area)

print("Total MAC cycles = %s" %mac_cycle)
print("Area of single 8-Bit MAC = %.2f um^2" %MAC_Single_Area)
print("Power consumed by a single 8-Bit MAC = %.2f nW" %Power_Consumed)
print('Total cycles required for operation: %s' %(cnt_single_mac))
```

Figure 4.11: Code snippet for CNN operation using single MAC block

Figure 4.11 presents the code block that is used to perform the convolution operation in a CNN. One MAC unit is used, which is the multiplication and addition operation within the loop. The loop makes sure that the multiplication of a $k \times k$ kernel is performed on the same size of an image portion and strides one pixel throughout the image until the whole convolution operation is performed. The accumulated result of the operation

is stored in the *outFS* which is a 2-D array. The block also computes the number of cycles required for the whole CNN operation to complete, along with the area and power consumption of the MAC unit.

Similar operation is performed by the code block in Figure 4.12 which performs the MAC operation considering the sparse input data. The difference being the if-else block within the loop that ensures that no zero inputs are considered and that is reflected by the smaller total number of cycles used to perform the whole convolution operation.

```
# operation block performing convolution operation using a single MAC considering sparse input data
outFSS_v = []

cnt_single_mac = 0
mac_cycle = 0
for strideY in range(stride):
    for strideX in range(strides):
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                if( IM == 0 and K):
                    pass
                else:
                    mul = IM * K
                    acc = acc + mul
                    cnt_single_mac += 1
        outFSS_v.append(acc)
        if(acc != 0):
            mac_cycle += 1
        #print(acc)
outFSS = array(outFSS_v)
outFSS = outFSS.reshape((int(outFSS.shape[0]/strides), strides))

print(outFSS)
print('Total MAC cycles: %s' %mac_cycle)
print('Total operation cycles: %s' %(cnt_single_mac))
```

Figure 4.12: Code snippet for CNN operation using single MAC block with sparse input

Similarly, Figure 4.13 and 4.14 represents the code blocks that computes output feature maps of a convolution operation using parallel MAC functions $MAC(in_1, k_1)$ and $MAC\_SP(in_1, k_1)$. These functions perform MAC operations using parallel MACs, with and without sparsity. The code blocks also calculates the area and power of the parallel MAC construction, and also computes the number of MACs along with the total operation cycles that can be used to compare performance. Additionally, the number of gates used are also computed which differ from a normal MAC architecture to that of one considering sparse data architecture.

```
# operation block performing convolution operation using a parallel MACs
cntCycle = 0
cnt = 0
cycle = 0
outFP_v = []

for strideY in range(stride):
    for strideX in range(strides):
        in_1 = np.array([])
        k_1 = np.array([])
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                in_1 = np.append(in_1, IM)
                k_1 = np.append(k_1, K)

        #print(in_1)
        #print(k_1)
        result = MAC(in_1, k_1)
        #print(result[0])
        outFP_v.append(result[0])

MACs_used = math.ceil(result[6]/result[1])
area = MAC_Area(MACs_used)
power = Power(area)

outFP = array(outFP_v)
outFP = outFP.reshape((int(outFP.shape[0]/strides), strides))

#print(outFP)
print('Total MAC cycles: %s' %result[1])
print('Total arithmatic cycles: %s' %result[6])
print('Total OR gates used: %s' %result[2])
print('Total XOR gates used: %s' %result[3])
print('Total And gates used: %s' %result[4])
print('Total arithmatic operations performed: %s' %result[5])
print('Total Area of MAC block: %.2f um^2' %area)
print('Total power consumed: %.2f nW' %power)
print('Total MACs used = %s' %MACs_used)
```

Figure 4.13: Code snippet for CNN operation using parallel MACs

```
# operation block performing convolution operation using parallel MACs considering sparse input data
cntCycle = 0
cnt = 0
outFPS_v = []
mac_cycle = 0

for strideY in range(stride):
    for strideX in range(strides):
        in_1 = []
        k_1 = []
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                in_1.append(IM)
                k_1.append(K)

        #print(in_1)
        #print(k_1)
        result = MAC_SP(in_1, k_1)
        outFPS_v.append(result[0])
        if(result[0] != 0):
            mac_cycle += 1

MACs_used = math.ceil(result[6]/mac_cycle)
area = MAC_Area(MACs_used)
power = Power(area)
outFPS = array(outFPS_v)
outFPS = outFPS.reshape((int(outFPS.shape[0]/strides), strides))

print('Total MAC cycles: %s' %mac_cycle)
print('Total arithmatic cycles: %s' %result[6])
print('Total OR gates used: %s' %result[2])
print('Total XOR gates used: %s' %result[3])
print('Total And gates used: %s' %result[4])
print('Total arithmatic operations performed: %s' %result[5])
print('Total Area of MAC block: %.2f um^2' %area)
print('Total power consumed: %.2f nW' %power)
print('Total MACs used: %s' %MACs_used)
#print(outFPS)
```

Figure 4.14: Code snippet for CNN operation using parallel MACs with sparse input

5 Analysis

This chapter discusses the relation of different parameters such as area, power, gate usage and performance of the MAC units. The focus is made on parallel MAC operation since it is more likely to affect on better performance. Both normal and sparse inputs were considered for analysis. After running the code blocks, it was seen that the sparsity of data using a threshold of $2^8 = 256$ provides is around 0.50; meaning that almost half of the values of 28×28 image input is zero. The analysis and discussion is done based on a MAC design with data size of 8-bit. For 16 and 32 bit word length the data would scale significantly and can be done with a few tweaks in the codes. For simplicity, an 8-bit MAC [MACs which takes two 8-bit inputs] is chosen and discussed. Analysis is done using graphical representation between relationships of the parameters. MATLAB was used to conduct the analysis for its powerful data analysis capabilities.

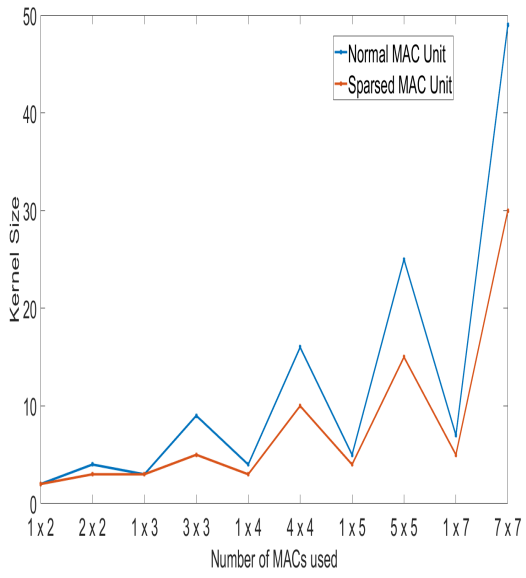


Figure 5.1: Graph showing the relationship between Number of MACs and Kernel size

Kernel Size	Total MAC	Total MAC (sparse)
1 x 2	2	2
2 x 2	4	3
1 x 3	3	3
3 x 3	9	5
1 x 4	4	3
4 x 4	16	10
1 x 5	5	4
5 x 5	25	15
1 x 7	7	5
7 x 7	49	30

Table 5.1: Data for total MACs and kernel size/dimension based on normal and sparse input

The number of MACs, based on kernel size or dimension, for both normal and sparse inputs are demonstrated in Table 5.1 and their trend is shown in Figure 5.1. It is clearly seen that the number of MACs for sparse inputs (red line) are less than that of normal

input data. It is also observed that the total number of MACs increases more significantly for a 2-D kernel than that of a 1-D kernel. This is expected as the number of multiply and addition operation increases as the size of the kernel increases. The rest of the sections in this chapter demonstrates the MAC numbers from the same kernel dimension.

5.1 MACs vs. Area

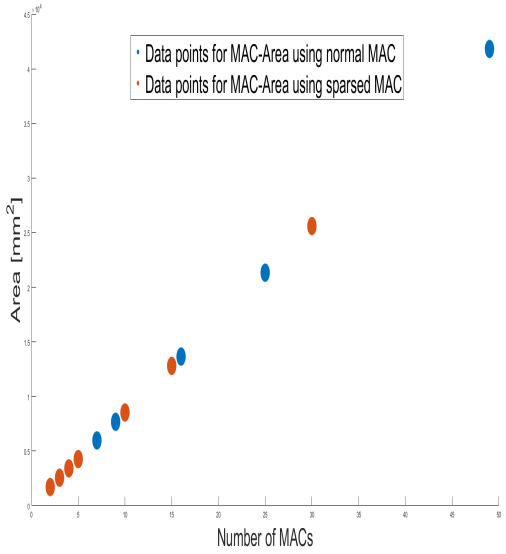


Figure 5.2: Scatter plot showing the relationship between Number of MACs and Area

Normal		Sparse	
Total MAC	**Area mm^2**	**Total MAC**	**Area mm^2**
2	1706.88	2	1706.88
4	3413.76	3	2560.32
3	2560.32	3	2560.32
9	7680.96	5	4267.20
4	3413.76	3	2560.32
16	13655.04	10	8534.40
5	4267.20	4	3413.76
25	21336.00	15	12801.60
7	5974.08	5	4267.20
49	41818.56	30	25603.20

Table 5.2: Data for total MAC and Area based on normal and sparse input

Figure 5.2 represents the scatter plot for the total number of MACs used in parallel and the Area. Respective data for the plot is presented in Table 5.2. Area increases with number of MACs as the number of gates increases which is reflected in the plot. It is observed that the total number of MAC reduces for the MAC designed with sparse input compared to that of normal input data. This is an expected outcome since the number of operations reduce for sparse input which also reduces the number of MACs, this ultimately reducing area. Another observation is that the number of Area increases linearly with the increase in number of MACs. The area of the fictional block will scale based on the number of blocks that run in parallel.

5.2 Performance

5.2.1 Number of MACs vs. Operation

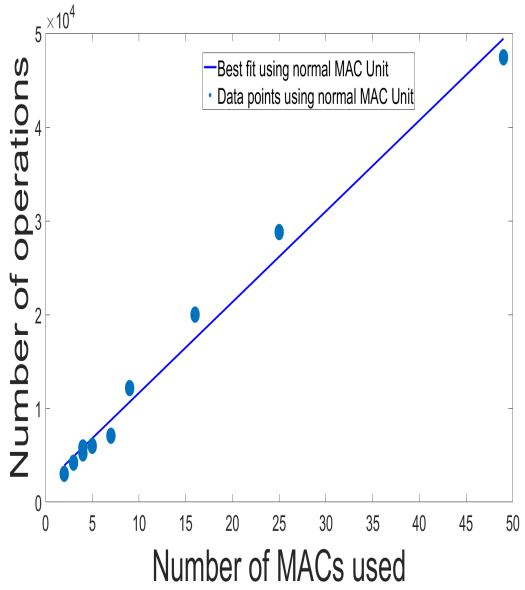


Figure 5.3: Scatter plot with best fit showing the relationship between Number of MACs and number of operations

Kernel Size	Total MACs	Total Operations
1 x 2	2	3024
2 x 2	4	5832
1 x 3	3	4212
3 x 3	9	12168
1 x 4	4	5200
4 x 4	16	20000
1 x 5	5	6000
5 x 5	25	28800
1 x 7	7	7084
7 x 7	49	47432

Table 5.3: Data for total MAC and Operations with kernel dimension using parallel Normal MAC units

Figure 5.3 showcases the relationship between number of MACs used in parallel and the respective total number of cycles required for the whole convolution operation. The relevant data are presented in Table 5.3 alongside the dimension of the kernel. Operations represent the number of cycles that the MACs use in total. Normal image inputs are considered using MACs in parallel to conduct the analysis. It can be observed that the number of MACs increases as the kernel dimension move from 1-D to 2-D. The total number of operations for 1-D kernel is less than that of 2-D kernel. This is expected as more calculations are involved when the the number of kernel values increases. The data points represent a scattered manner for which a scatter plot is chosen and its best fit line is drawn to provide the linear relationship of data.

5.2.2 Number of MACs vs. Operation for sparse input

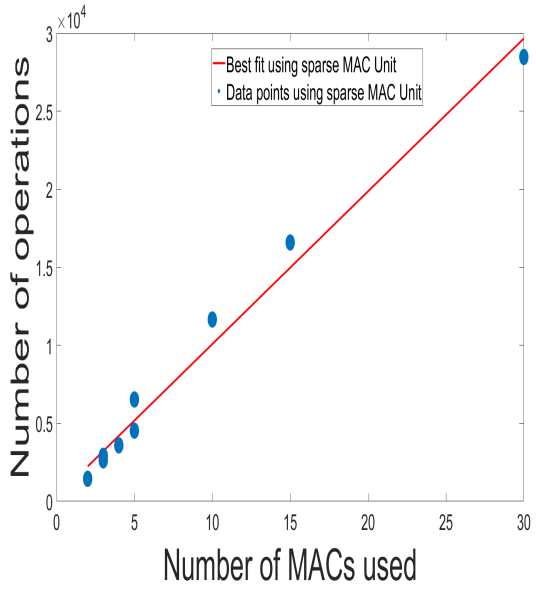


Figure 5.4: Scatter plot with best fit showing the relationship between Number of MACs and number of operations considering sparse inputs

Kernel Size	Total MACs	Total Operations
1 x 2	2	1444
2 x 2	3	2918
1 x 3	3	2810
3 x 3	5	6528
1 x 4	3	2624
4 x 4	10	11652
1 x 5	4	3584
5 x 5	15	16586
1 x 7	5	4528
7 x 7	30	28474

Table 5.4: Data for total MAC and Operations with kernel dimension using parallel MAC units considering sparse inputs

A similar trend in data can be seen in Figure 5.4 and Table 5.4 for parallel MAC operation considering sparse inputs. The significant difference of data in Table 5.3 and 5.4 is that less number of cycles are required when a spars input is considered, Since the MACs do not take the zero input values into consideration, less number of operations are performed which results in less cycles being required to complete the whole convolution operation. The difference is between the two types of MAC construction is more evident in Figure 5.5. The red line shows the sparse MAC operation and the blue line represent normal operation.

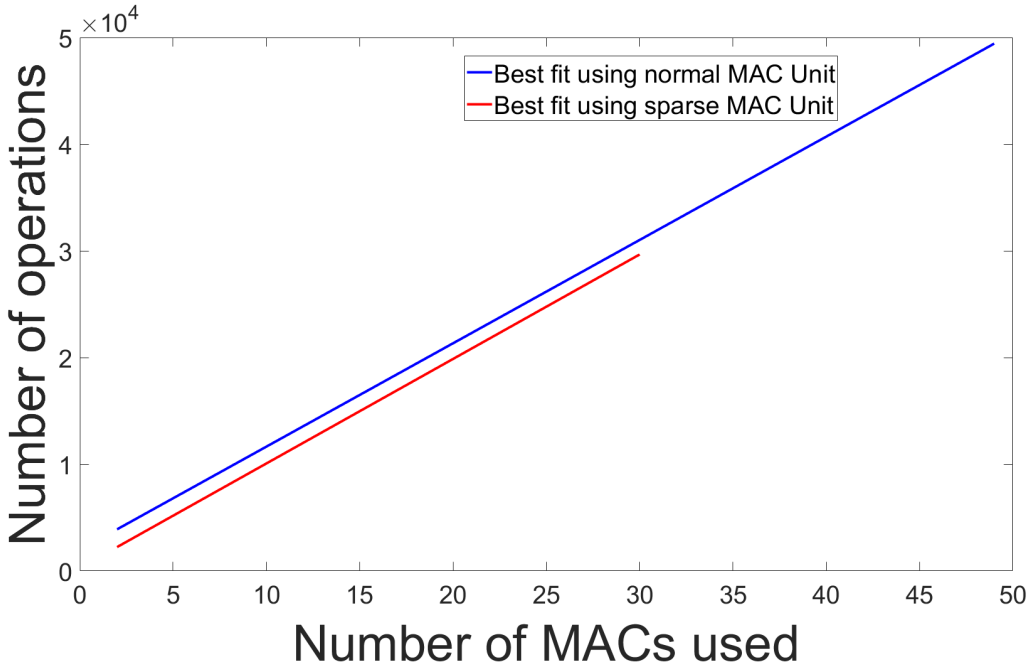


Figure 5.5: Graphical presentation of the two best it lines for MAC operations in parallel.

5.3 Gate Usage

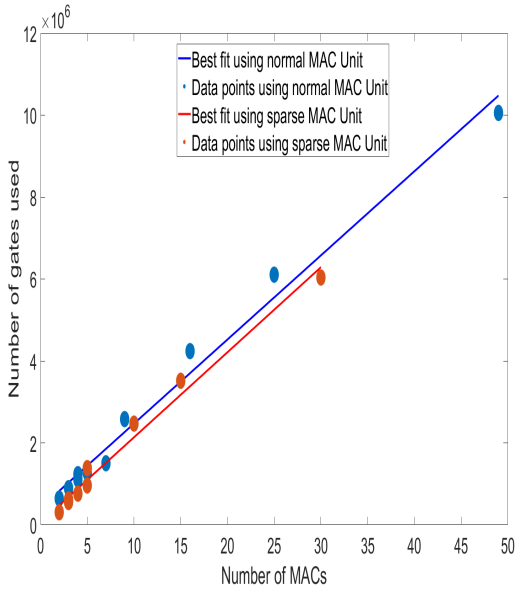


Figure 5.6: Scatter plot with best fit showing the relationship between Number of MACs and number of gates

Total MAC	Total Gate	Total MAC Sparse	Total Gate Sparse
2	641088	2	306128
4	1236384	3	618616
3	892944	3	595720
9	2579616	5	1383936
4	1102400	3	556288
16	4240000	10	2470224
5	1272000	4	759808
25	6105600	15	3516232
7	1501808	5	959936
49	10055584	30	6036488

Table 5.5: Data for total MAC and Gates using parallel MAC units

Figure 5.6 and Table 5.5 presents the relationship between the total number of gate operation with the respective number of MACs used in parallel. The gates include combination of AND, OR and XOR gates that are used for the MAC construction. The number of gates presented Table 5.5 does not mean the presence of that many gates, rather the number of times the gates were used in combination when the MAC blocks are in operation. It is observed that the number of MACs reduces when sparse input is considered. This is expected as less number of operation means less usage of gates and this is supported by the in Table 5.4. The best fit lines shows the trend in data points for MACs considering with and without sparse input.

5.4 Power

5.4.1 Area Vs. Power

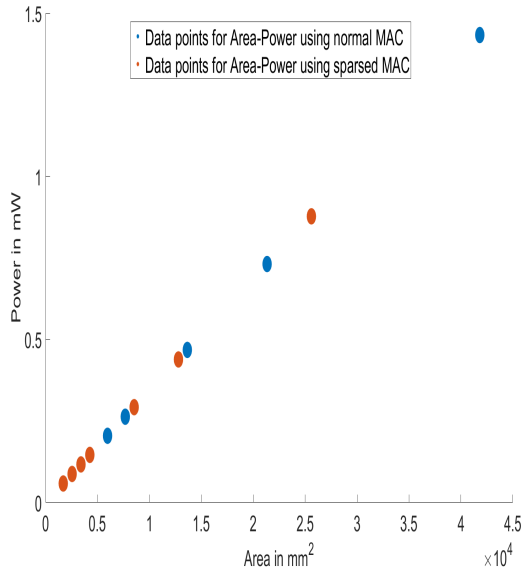


Figure 5.7: Scatter plot showing the relationship between Area and Power

Area [mm^2]	Power [mW]	Area sparse [mm^2]	Power sparse [mW]
1706.88	0.05849819	1706.88	0.05849819
3413.76	0.11699638	2560.32	0.08774729
2560.32	0.08774729	2560.32	0.08774729
7680.96	0.26324186	4267.20	0.14624548
3413.76	0.11699638	2560.32	0.08774729
13655.04	0.46798553	8534.40	0.29249096
4267.20	0.14624548	3413.76	0.11699638
21336.00	0.73122739	12801.60	0.43873644
5974.08	0.20474367	4267.20	0.14624548
41818.56	1.43320569	25603.20	0.87747287

Table 5.6: Data for Area and Power using parallel MAC units

Figure 5.7 shows the linear relationship between Area and Power. It is an expected relation as the power consumption directly depends on the size of the component. Bigger unit consumes more power and the figure suggests the same pattern. As number of MACs

are used more the overall size or area of the architecture increases, this also increases the power significantly as is seen in Table 5.6. The power is significantly seen to be less for MACs that considers sparse input than that of normal input. This is supported by the largest value shown in the table, which is comparatively significant in value.

5.4.2 MAC Vs. Power

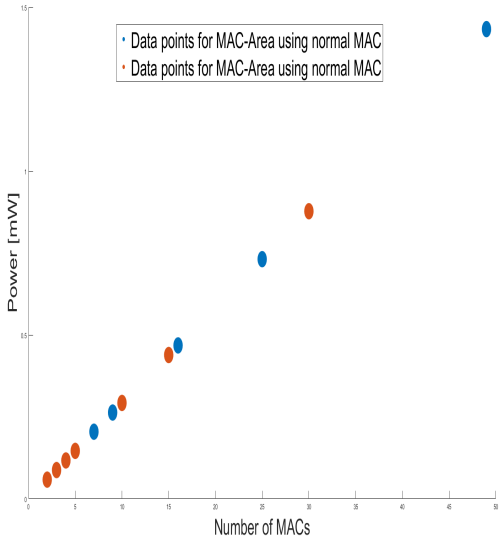


Figure 5.8: Scatter plot showing the relationship between number of MACs and Power

Total MAC	Power [mW]	Total MAC sparse	Power sparse [mW]
2	0.05849819	2	0.05849819
4	0.11699638	3	0.08774729
3	0.08774729	3	0.08774729
9	0.26324186	5	0.14624548
4	0.11699638	3	0.08774729
16	0.46798553	10	0.29249096
5	0.14624548	4	0.11699638
25	0.73122739	15	0.43873644
7	0.20474367	5	0.14624548
49	1.43320569	30	0.87747287

Table 5.7: Data for Power using parallel MAC units with and without sparse input

Similar trend in seen in Figure 5.8 and Table 5.7 as discussed in the section 5.4.1. Figure 5.8 supports Figure 5.8 which proves that as more MACs are used, the power consumption is also more.

6 Discussion

6.1 Design

Considering the objective of the thesis work, the analytical model was designed on high level using Python. Python makes designing of AI models easy which is the main reason for choosing this language. Functional code blocks are written to perform the analysis. Using functional blocks makes it way to maintain and re-factor.
The design that is used in this study consider a 8-bit word length for simplicity. The same design can a tweaked to accommodate 16-bit and 32-bit word length and perform the same analysis. The most significant affect by using a 16-bit or 32-bit input will be on the *area* and *power*. Also, the kernel dimensions used to obtain the results range from 2-7. This is because the standard kernel used in CNN operation is usually $2\times2, 3\times3, 5\times5 or 7\times7$[17].

Sparse inputs are considered as an alternative method to that of a traditional one. It is observed that for most of the calculations in convolution operation, the zero input values serves no purpose in calculating the output feature map. These redundant calculations just increases computation cycles which is power hungry. Also the threshold value can be used as a parameter to control the input matrix and remove all the redundant data. For instance, in a radio application, if the design is used to identify signal, then the threshold value can be used to eliminate the signal values from the input. In hardware this operation can be controlled by a comparator unit/block that could filter out the values. Considering sparse inputs has shown significant reduction in MAC operation which results in less area and power consumption.
The kernel size has been modified to fit both 1-D and 2-D representation of kernel/weight data. With a few minor modification in the codes, the design can be accommodated to calculate output for 1-D input image as well. This might be more helpful to analyze and design an AI accelerator that deals with 1-D input, say a time series data.

6.2 Area

Area calculation is done based on the 8-bit word length of the image and kernel inputs. This represents a simpler design which scales up significantly as the word length of the inputs increases to 16-bit and 32-bit. For best implementation, 32-bit inputs are considered can be implemented by further tweaking the code. This can be considered as a future scope of study.

If a 16-bit and 32-bit word length is considered for the inputs, the area would also be increased. On the same axis, the line graphs for the two will lie on top of the data points that are presented in Figure 5.2. The graphical representations in section 5.1 provides an idea of relationship between area and total MACs that can be used in parallel. The data points could be used to draw a best fit line which will show a linear relationship between the two parameters. The line can be used to get an approximation of the total number of MACs that can be implemented with a given size/area and vice-versa. The area calculation in this study is an approximate value for a real-life implementation, so it may be used as a reference to have an approximate idea of the number of MACs that can be fit in a System on Chip (SoC) for a given area.

6.3 Power

Similar idea to obtain an assumption for area can be used for power consumption as well. Since power consumption is directly related to area, an increase in area would also cause a linear increase in power; as is supported by Figure 5.7. There is also the obvious conclusion that considering sparsity in the input data helps in significant reduction of power consumption, and the same is demonstrated in Table 5.6.

The power calculation is done keeping the dynamic/active power of transistors in mind. There are other factors that are taken into account for total power. Since active power is the most controlling factor of power consumption, it was considered in this study to provide value for approximate power. The model derived in this study for power consumption gives a generic idea of the power consumption, and can be used for identifying power with given number of MACs and area.

6.4 Performance

The performance measure in this study is the number of clock cycles and MAC operations for completing a whole convolution. Performance is directly related to the number of MACs that are used in total. According to the results, a single MAC with 8-bit 28×28 image and 7×7 kernel requires 3388 number of MAC operations. Whereas using 7-MACs the number of operations seen was 484. This proves that increasing number of MACs to run in parallel increases the performance as well, with the trade-off in power consumption and area.

7 Conclusion

The idea of this thesis is to design an analytical model for MAC architecture and obtain performance, area and power from the design. This was successfully implemented and the results were found as expected. Since this is an anlytical study, the values obtained from this model can be used as a reference to design and implement actual MAC accelerator in hardware that can perform AI and more specifically CNN operation.

The values obtained from the result of this study can be a helpful insight for digital designers for design consideration. The result might not be exact but is a close approximation as the model is designed keeping hardware design in mind. The analysis of this report helps identify the relationship that one can expect while designing MAC units for area, power and performance. Since all the relationships are linear, hardware designers can have an insight of the maximum number of MACs that can be fit into an AI accelerator for a given size and also have a suggested power consumption.

In future, the model can be extended to fit all the activities, including activation, pooling and flattening within an entire CNN operation. Moreover, the software implementation gives an idea on how to implement the same in a hardware. This study is an entry point to a much bigger study and can be extended to fit diverse AI applications.

References

[1] J. D. Peters, "Convolutional neural network." In *https://goo.gl/images/SWsgqh*.

[2] F. Doukkali, "Convolutional neural networks (cnn, or convnets)." In *https://medium.com/@phidaouss/convolutional-neural-networks-cnn-or-convnets-d7c688b0a207*. [Online; accessed June, 2019.].

[3] E. Tutorial, "A 4-bit ripple carry adder." In *https://www.electronics-tutorials.ws/combination/comb_7.html*. [Online; accessed June, 2019.].

[4] I. Amir, "4 bit binary full adder – logic gate." In *https://izatxamir.wordpress.com/2010/02/01/4-bit-binary-full-adder-logic-gate/*. [Online; accessed June, 2019.].

[5] S. Hwang, "Convolutional neural network (cnn) presentation from theory to code in theano." In *https://www.slideshare.net/SeongwonHwang/convolutional-neural-network-cnn-presentation-from-theory-to-code-in-theano*. [Online; accessed June, 2019.].

[6] J. Garland and D. Gregg, "Low Complexity Multiply-Accumulate Units for Convolutional Neural Networks with Weight-Sharing," *arXiv e-prints*, p. arXiv:1801.10219, Jan 2018.

[7] L. Ranganath, D. Jay Kumar, and P. Siva Nagendra Reddy, "Design of MAC Unit in Artificial Neural Network Architecture using Verilog HDL," [Online; accessed June, 2019.].

[8] A. L. Samuel, "Some studies in machine learning using the game of checkers.," 1959.

[9] T. M. Mitchell, *Machine learning*. McGraw-Hill Science/Engineering/Math, March, 1997.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] Stanford., "Convolutional Neural Networks for Visual Recognition.." *http://cs231n.github.io*. [Online; accessed May, 2017.].

[12] M. Nielsen, "Neural Networks and Deep Learning.." *http://neuralnetworksanddeeplearning.com/index.html*. [Online; accessed May, 2017.].

[13] Wikipedia, "Multiply–accumulate operation." In *https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation*. [Online; accessed June, 2019.].

[14] P. Siva Nagendra Reddy and A. G. Murali Krishna, "Implementation of RISC Processor for Convolution Applications," *International journal of Computer Trends and Technology*, vol. 4, pp. ISSN: 2231–2803(IJCTT), June 2013.

[15] R. Naresh Naik, P. Siva Nagendra Reddy, and K. Madan Mohan, "Design of Vedic Multiplier for Digital Signal Processing Applications," *International Journal of Engineering Trends and Technology*, vol. 4, pp. ISSN: 2231–5381(IJETT), July 2013.

[16] Wikibooks, "Vedic mathematics/techniques/multiplication." In *https://en.wikibooks.org/wiki/Vedic_Mathematics/Techniques/Multiplication*. [Online; accessed June, 2019.].

[17] S. Sahoo, "Deciding optimal kernel size for cnn." In *https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363*. [Online; accessed June, 2019.].

Appendix

A1 Code

```
#---------------- Library imports ---------------------------
import numpy as np
import math
from numpy import array, reshape, count_nonzero
from collections import namedtuple
#----------------- Variable declaration ----------------------
dimIR = 28 #row-dimension of input image matrix
dimIC = 28 #column-dimension of input image matrix
dimWR = 5 #row-dimension of kernel/weight matrix
dimWC = 5 #column-dimension of input kernel/weight matrix
cnt = 0 # for counting the number of MAC operations
cnt_sp = 0 # for counting the number of sparsed MAC operations
cntCycle = 0 # for getting the number of cycles within one MAC operation
    ↪ | initialized to zero whenever MAC function is called
cycle = 0 # for geting the number of MAC operations based on strides
    ↪ without cosidering sparsed input data
cycle_SM = 0 # for geting the number of MAC operations based on strides
    ↪ cosidering sparsed input data


#------- Creating the input matrix ---------------------------
iM = np.random.randint(0, 500, size = (dimIR, dimIC)) # random generation
    ↪  of input image matrix
kM = np.random.randint(-1, 5, size = (dimWR, dimWC)) # random generation
    ↪ of kernel matrix


# printing for debugging purposes ---
#print(iM)
#print(kM)
```

```
#-----------------------------------

# calculating the number of stride for CNN operation based on the
    ↪ dimension of image and kernel
strides = (dimIC - dimWC) + 1

# logic to handle stride value based on kernel dimention to fit 1-D/2-D
    ↪ array
if (dimWC != dimWR):
    stride = strides + 1 # stride value for 1-D kernel size
else:
    stride = strides # stride value for 2-D kernel size



# In[13]:



bitSwap = 8 # variable to control numbber of gates | 8 for 8-bit
    ↪ representation and 6 for 16-bit representation

# Code block for logic gate number and area representation
#--- Size in um^2 for logic gates as per Nordic library ------
And = 1.68 # size for 2 input And gate
Or = 1.40 # size for 2 input Or gate
Xor = 2.80 # size for 2 input XOr gate

#-------- Parameters for calculating power -------------------
voltage = 1.2 # as per 55nm technology
frequency = 100 #100MHZ
activity_factor = 0.2 #20% activity factor
capacitance = 4.04 #nF
```

```
#-------- Parameters of logic gates used in MAC --------------
# considering a 8-bit MAC
orGate_8 = 72
xorGate_8 = 144
andGate_8 = 208

# considering a 16-bit MAC
orGate_16 = 272
xorGate_16 = 544
andGate_16 = 800

if (bitSwap == 16):
    totalORGates = orGate_16
    totalXORGates = xorGate_16
    totalANDGates = andGate_16
else:
    totalORGates = orGate_8
    totalXORGates = xorGate_8
    totalANDGates = andGate_8

#print(totalORGates, totalXORGates, totalANDGates)

# Mac struct to define the design of MAC using logic gates
MacStruct = namedtuple("MacStruct", "OR XOR AND")

# variable declaring total number of gates for one n-bit MAC
gate = MacStruct(totalORGates, totalXORGates, totalANDGates)



# In[14]:
```

```
# Function for Calculating Area
def MAC_Area(MACs):
    Area = (totalANDGates * And + totalORGates * Or + totalXORGates * Xor
        ↪ ) * MACs
    return Area


# In[15]:


# Function for calculating power considering area
def Power(Area):
    global voltage, frequency, activity_factor, capacitance
    power = 0.5 * activity_factor * capacitance * (voltage * voltage) *
        ↪ Area
    return int(power)


# In[16]:


# Parallel MAC functional block conducting MAC operation of the same size
    ↪  as kernel dimension
def MAC(a,b):
    global acc, cnt, cycle, cntCycle
    global And, Or, Xor

    for item in range(len(a)):
        mul = a[item]*b[item]
        cntCycle += 1
        acc = acc + mul
        cntCycle += 1
```

```
        cnt +=1 #getting the mac operation considering each value
    cycle += 1 #getting the number of cycles for whole operation based on
        ↪ kernel size

    OrGate = gate.OR * cnt
    XorGate = gate.XOR * cnt
    AndGate = gate.AND * cnt

    return (acc, cycle, OrGate, XorGate, AndGate, cntCycle, cnt)


# In[17]:


# Parallel MAC operation fucntion considering sparsed input
def MAC_SP(im_in,k_in):
    global acc, cnt_sp, cycle_SM, cntCycle
    global And, Or, Xor

    for item in range(len(im_in)):
        if (im_in[item] == 0 and k_in[item]):
            pass
        else:
            mul = im_in[item]*k_in[item]
            cntCycle += 1
            acc = acc + mul
            cntCycle += 1
            cnt_sp += 1
        cycle_SM += 1

    OrGate = gate.OR * cnt_sp
    XorGate = gate.XOR * cnt_sp
```

```
    AndGate = gate.AND * cnt_sp

    return (acc, cycle_SM, OrGate, XorGate, AndGate, cntCycle, cnt_sp)


# In[18]:


cnt_single_mac = 0
outFS_v = []
mac_cycle = 0

for strideY in range(stride):
    for strideX in range(strides):
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                #print(IM, K)
                mul = IM * K
                acc = acc + mul
                cnt_single_mac += 1
            mac_cycle += 1
        outFS_v.append(acc)
outFS = array(outFS_v)
outFS = outFS.reshape((int(outFS.shape[0]/strides), strides))
#print(outFS)
MAC_Single_Area = MAC_Area(1)
Power_Consumed = Power(MAC_Single_Area)

print("Total MAC cycles = %s" %mac_cycle)
```

```
print("Area of single 8-Bit MAC = %.2f um^2" %MAC_Single_Area)
print("Power consumed by a single 8-Bit MAC = %.2f nW" %Power_Consumed)
print('Total cycles required for operation: %s' %(cnt_single_mac))



# In[19]:



cntCycle = 0
cnt = 0
cycle = 0
outFP_v = []

for strideY in range(stride):
    for strideX in range(strides):
        in_1 = np.array([])
        k_1 = np.array([])
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                in_1 = np.append(in_1, IM)
                k_1 = np.append(k_1, K)

        #print(in_1)
        #print(k_1)
        result = MAC(in_1, k_1)
        #print(result[0])
        outFP_v.append(result[0])

MACs_used = math.ceil(result[6]/result[1])
```

```
area = MAC_Area(MACs_used)
power = Power(area)

outFP = array(outFP_v)
outFP = outFP.reshape((int(outFP.shape[0]/strides), strides))

#print(outFP)
print('Total MAC cycles: %s' %result[1])
print('Total arithmatic cycles: %s' %result[6])
print('Total OR gates used: %s' %result[2])
print('Total XOR gates used: %s' %result[3])
print('Total And gates used: %s' %result[4])
print('Total arithmatic operations performed: %s' %result[5])
print('Total Area of MAC block: %.2f um^2' %area)
print('Total power consumed: %.2f nW' %power)
print('Total MACs used = %s' %MACs_used)


# In[20]:



iM[iM > 256] = 0
cnt = 0
print(iM)
print(kM)
# calculate sparsity
sparsity = 1.0 - (count_nonzero(iM) / iM.size)
print("Sparsity of input data = %s" %sparsity)



# In[21]:
```

```
outFSS_v = []

cnt_single_mac = 0
mac_cycle = 0
for strideY in range(stride):
    for strideX in range(strides):
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                if( IM == 0 and K):
                    pass
                else:
                    mul = IM * K
                    acc = acc + mul
                    cnt_single_mac += 1
        outFSS_v.append(acc)
        if(acc != 0):
            mac_cycle += 1
        #print(acc)
outFSS = array(outFSS_v)
outFSS = outFSS.reshape((int(outFSS.shape[0]/strides), strides))

#print(outFSS)
print('Total MAC cycles: %s' %mac_cycle)
print('Total operation cycles: %s' %(cnt_single_mac))



# In[22]:
```

```
cntCycle = 0
cnt = 0
outFPS_v = []
mac_cycle = 0

for strideY in range(stride):
    for strideX in range(strides):
        in_1 = []
        k_1 = []
        acc = 0
        for iRow in range(dimWR):
            for iCol in range(dimWC):
                IM = iM[iRow+strideY][iCol+strideX]
                K = kM[iRow][iCol]
                in_1.append(IM)
                k_1.append(K)

        #print(in_1)
        #print(k_1)
        result = MAC_SP(in_1, k_1)
        outFPS_v.append(result[0])
        if(result[0] != 0):
            mac_cycle += 1

MACs_used = math.ceil(result[6]/mac_cycle)
area = MAC_Area(MACs_used)
power = Power(area)
outFPS = array(outFPS_v)
outFPS = outFPS.reshape((int(outFPS.shape[0]/strides), strides))

print('Total MAC cycles: %s' %mac_cycle)
```

```
print('Total arithmatic cycles: %s' %result[6])
print('Total OR gates used: %s' %result[2])
print('Total XOR gates used: %s' %result[3])
print('Total And gates used: %s' %result[4])
print('Total arithmatic operations performed: %s' %result[5])
print('Total Area of MAC block: %.2f um^2' %area)
print('Total power consumed: %.2f nW' %power)
print('Total MACs used: %s' %MACs_used)
#print(outFPS)
```

A2 Parameter Values

For power calculation, the following values were obtained for a typical signal processing unit within Nordic Semiconductor's 55nm technology library:

- Average Current, $I_{avg} = 60mA$
- Operating Voltage, $V = 1.1V$
- Total number of gates, $G = 1.7M$
- Operating frequency, $f = 180MHz$
- Activity factor, $\alpha = 15\%$

These values were used to calculate active power of a gate using $P = V \times I$ and hence calculate the capacitance using the following formula:

$$P = \frac{1}{2} \times Activity factor \times Capacitance \times Operating Voltage \times Operating frequency \quad (.1)$$

For this thesis work, the following parameters were considered to calculate dynamic power:

- Operating frequency, $f = 100MHz$
- Operating Voltage, $V = 1.2V$, as per Nordic's 55nm library
- Activity factor, $\alpha = 20\%$

A3 Result Data

	Single MAC								
	Normal MAC					Sparsed MAC			
Kernel Size	Mac Number	Area [um^2]	MAC cycles	Operation Cycles	Power [nW]	Mac Number	Area [um^2]	MAC cycles	Operation Cycles
1 x 2	1	853.44	756	3024	29249.10	1	853.44	541	1444
2 x 2	1	853.44	1458	5832	29249.10	1	853.44	681	2918
1 x 3	1	853.44	702	4212	29249.10	1	853.44	521	2810
3 x 3	1	853.44	2028	12168	29249.10	1	853.44	669	6528
1 x 4	1	853.44	650	5200	29249.10	1	853.44	617	2624
4 x 4	1	853.44	2500	20000	29249.10	1	853.44	625	11652
1 x 5	1	853.44	600	6000	29249.10	1	853.44	532	3584
5 x 5	1	853.44	2880	28800	29249.10	1	853.44	576	16586
1 x 7	1	853.44	506	7084	29249.10	1	853.44	493	4528
7 x 7	1	853.44	3388	47432	29249.10	1	853.44	484	28474

Figure A3.1: Data from code using single MAC unit

				Parallel MAC						
				Normal MAC						
Mac Number	Area [um^2]	MAC cycles	Operation Cycles	Total OR Gates	Total XOR Gates	Total AND Gates	Gates Used	Power [nW]	Power [mW]	Power [nW] using gates
2	1706.88	756	3024	108864	217728	314496	641088	58498.19	0.05849819	24361344
4	3413.76	729	5832	209952	419904	606528	1236384	116996.38	0.11699638	46982592
3	2560.32	702	4212	151632	303264	438048	892944	87747.29	0.08774729	33931872
9	7680.96	676	12168	438048	876096	1265472	2579616	263241.86	0.26324186	98025408
4	3413.76	650	5200	187200	374400	540800	1102400	116996.38	0.11699638	41891200
16	13655.04	625	20000	720000	1440000	2080000	4240000	467985.53	0.46798553	161120000
5	4267.20	600	6000	216000	432000	624000	1272000	146245.48	0.14624548	48336000
25	21336.00	576	28800	1036800	2073600	2995200	6105600	731227.39	0.73122739	232012800
7	5974.08	506	7084	255024	510048	736736	1501808	204743.67	0.20474367	57068704
49	41818.56	484	47432	1707552	3415104	4932928	10055584	1433205.69	1.43320569	382112192

Figure A3.2: Data from code using parallel MAC unit for normal input

				Parallel MAC								
				Sparsed MAC								
Mac Number	Area [um^2]	MAC cycles	Operation Cycles	Total OR Gates	Total XOR Gates	Total AND Gates	Gates Used	Power [nW]	Power [mW]	Power [nW] using gates	Sparsity	Power Diff
2	1706.88	541	1444	51984	103968	150176	306128	58498.19	0.05849819	11632864	0.52	12728480
3	2560.32	681	2918	105048	210096	303472	618616	87747.29	0.08774729	23507408	0.50	23475184
3	2560.32	521	2810	101160	202320	292240	595720	87747.29	0.08774729	22637360	0.50	11294512
5	4267.20	669	6528	235008	470016	678912	1383936	146245.48	0.14624548	52589568	0.52	45435840
3	2560.32	617	2624	94464	188928	272896	556288	87747.29	0.08774729	21138944	0.50	20752256
10	8534.40	625	11652	419472	838944	1211808	2470224	292490.96	0.29249096	93868512	0.50	67251488
4	3413.76	532	3584	129024	258048	372736	759808	116996.38	0.11699638	28872704	0.50	19463296
15	12801.60	576	16586	597096	1194192	1724944	3516232	438736.44	0.43873644	133616816	0.49	98395984
5	4267.20	493	4528	163008	326016	470912	959936	146245.48	0.14624548	36477568	0.49	20591136
30	25603.20	484	28474	1025064	2050128	2961296	6036488	877472.87	0.87747287	229386544	0.48	152725648

Figure A3.3: Data from code using parallel MAC unit for sparse input

A4 Graphs

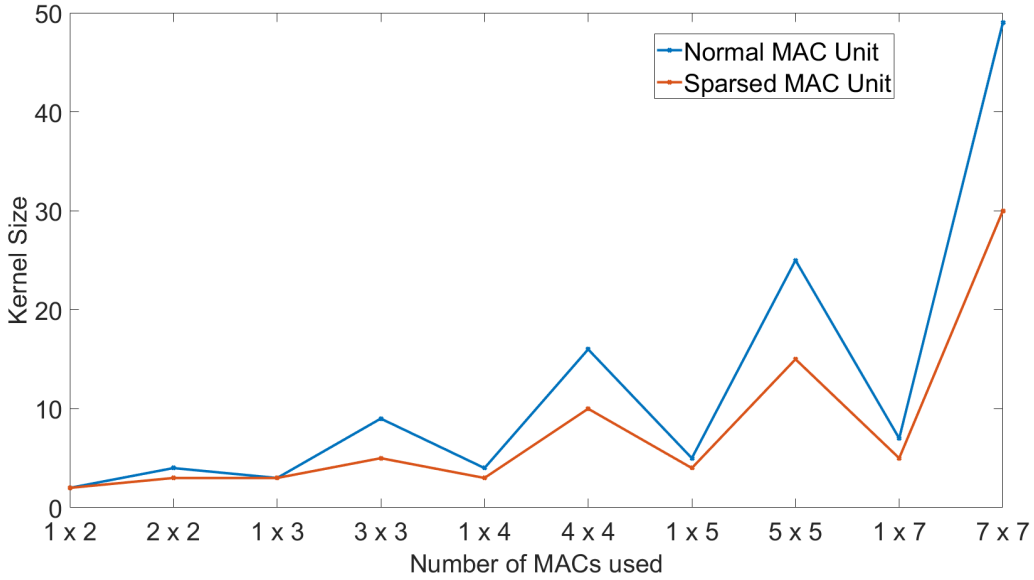


Figure A4.1: Graph showing the relationship between Number of MACs and Kernel size

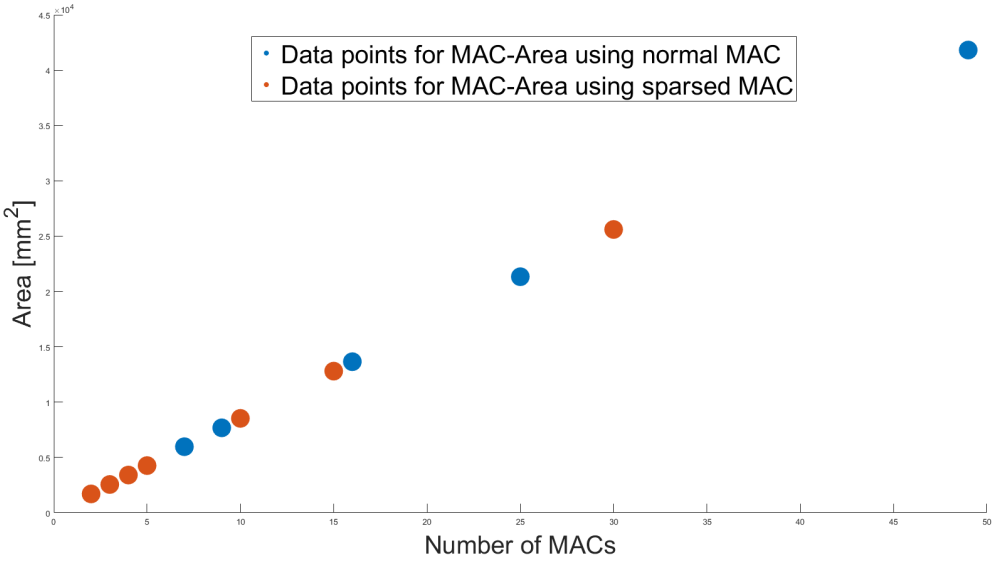


Figure A4.2: Scatter plot showing the relationship between Number of MACs and Area

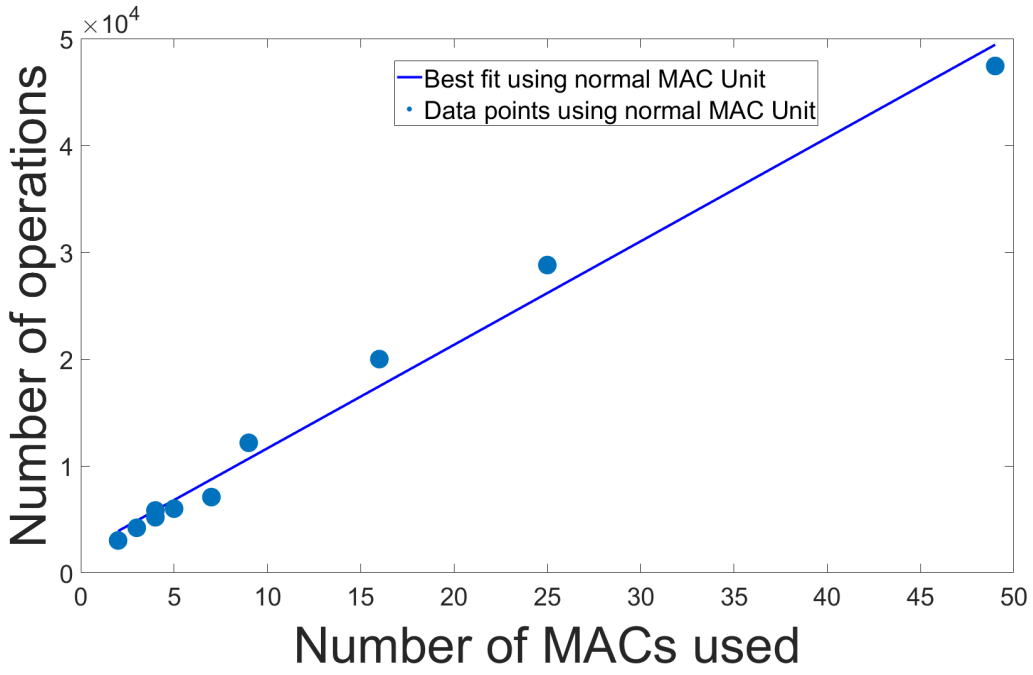


Figure A4.3: Scatter plot with best fit showing the relationship between Number of MACs and number of operations

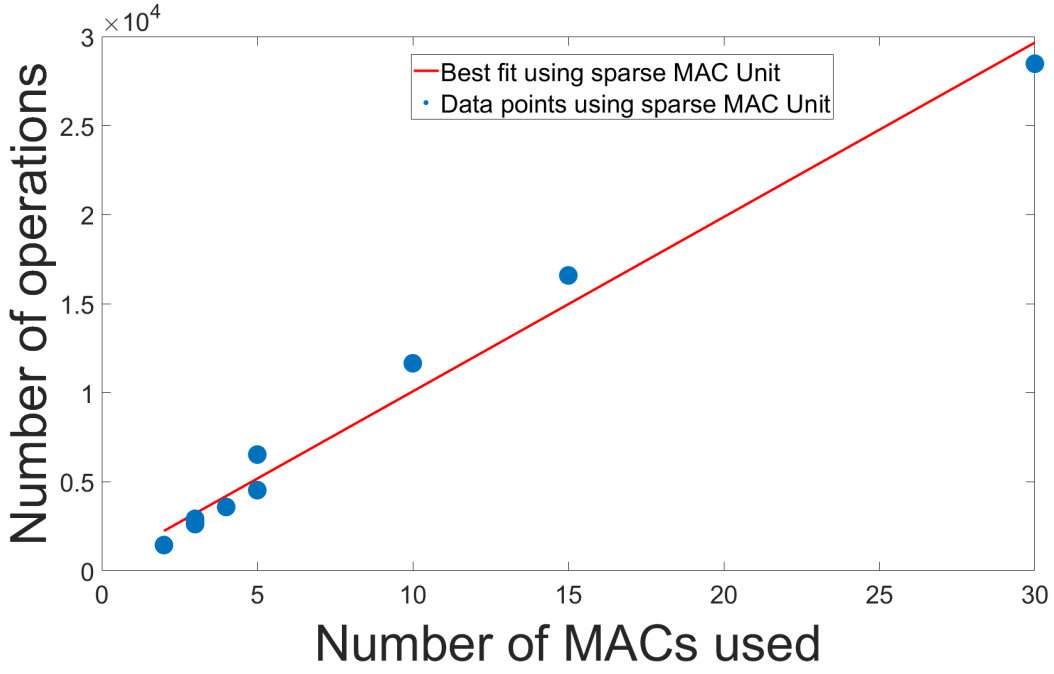


Figure A4.4: Scatter plot with best fit showing the relationship between Number of MACs and number of operations considering sparse inputs

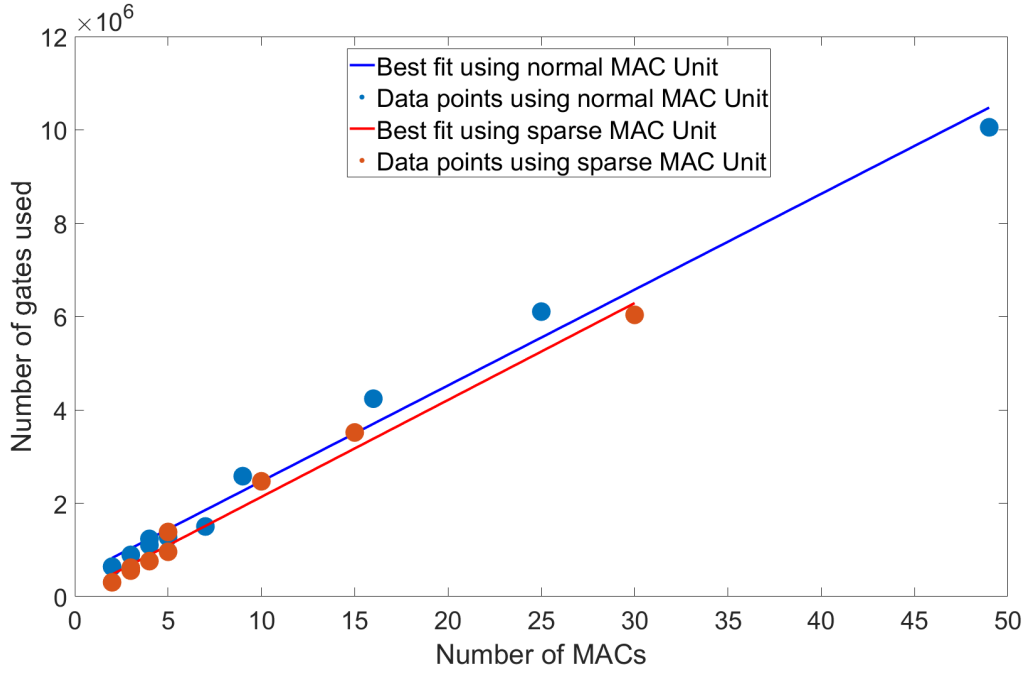


Figure A4.5: Scatter plot with best fit showing the relationship between Number of MACs and number of gates

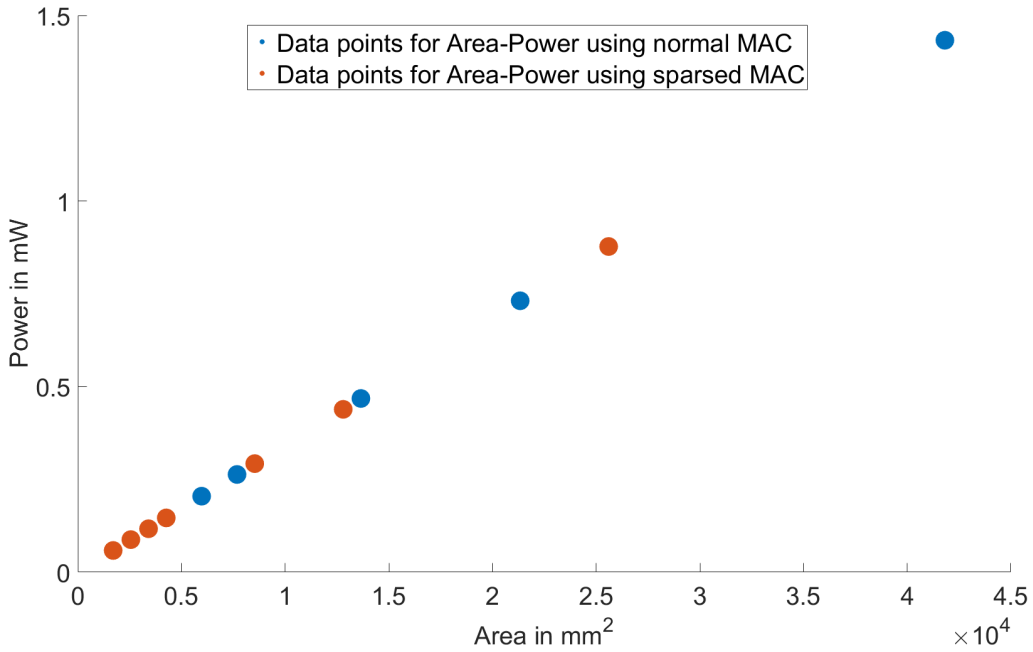


Figure A4.6: Scatter plot showing the relationship between Area and Power

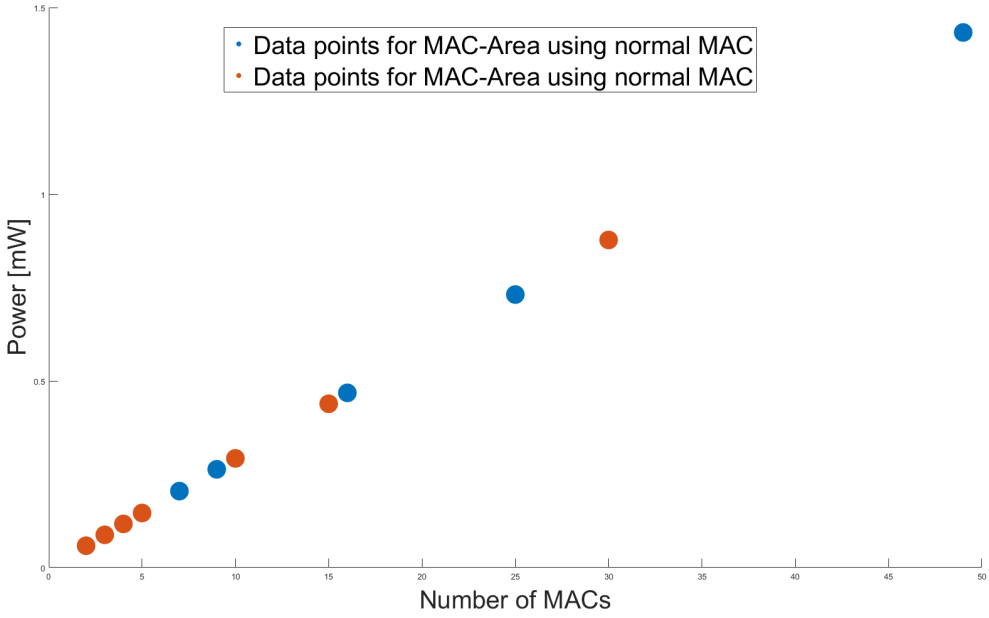


Figure A4.7: Scatter plot showing the relationship between number of MACs and Power