Tampere University

Samaneh Ammari

NEURAL NETWORK ACCELERATOR DESIGN FOR SYSTEM ON CHIP

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Timo D. Hämäläinen, Saman Payvar, M.Sc.
October 2022

ABSTRACT

Samaneh Ammari: Neural Network Accelerator Design for System on Chip
Master of Science Thesis
Tampere University
Master's degree
October 2022

ML is vastly utilized in a variety of applications such as voice recognition, computer vision, image classification, object detection, and plenty of other use cases. Protecting data privacy and the importance of preventing latency in different applications and saving the network bandwidth to process data locally without the need to transfer it to the cloud. The approach is called edge computing. It is challenging to design a deep learning accelerator suitable for edge devices. Two main factors affect the chip design. On-chip memory is the first and the most power and area-consuming unit. The second one is multipliers. In this thesis, we are focusing on the latter.

Most machine learning algorithms use convolution, which is calculated by multiplying and accumulating input feature maps and weights. Most of the deep learning accelerators use the precision scalable Multiply and Accumulate (MAC) architecture and an array of MAC units. Most of the chip's area is taken up by the array of MAC units, especially multipliers, which also use a lot of power.

This master's thesis consists of two parts. First, a new deep learning accelerator architecture is proposed. Second, different multiplier algorithms are explored. These algorithms were implemented in the SystemVerilog language and synthesized via Cadence tools. The aim was to find a smaller area and lower power consumption multiplier with higher performance. In this work, the Braun multiplier, the Booth multiplier, the Baugh-Wooley array multiplier, the Wallace multiplier, the Parallel prefix Vedic multiplier, and the Modified-Booth multiplier are implemented. The power consumption, chip area usage, and performance of the multipliers at different clock frequencies are measured and considered to select the optimal multiplier. Then the precision flexibility feature is added to the selected multiplier algorithms to perform one 8-bit*8-bit, two 4-bit*4-bit, or four 2-bit*2-bit multiplication. It is worth mentioning that both data (multiplicand) and weight (multiplier) can be in different bit width ranges, such as 1,2,4,8. In the proposed deep learning accelerator, the area and power of the systolic array are measured and reported. Among all other multipliers, the signed flexible Modified Booth multiplier which can calculate 2,4, and 8-bits is selected. It occupies 866.461 um^2, and consumes 0.653 mW power at 1 GHz. The area and power of the systolic array with bit precision flexible are 283,564 mm^2 and 223,156 mW power at 1 GHz, respectively.

Keywords: System on Chip, Embedded Systems, FPGA, ASIC, Machine Learning, Deep learning, Neural Network, Hardware accelerator, Deep learning accelerator

PREFACE

This work was done by a collaboration of Nokia Networks, and SoC-HUB, directed by Tampere University. It was my honor to be a part of this collaboration and work with wonderful people. I would like to thank all the people around me who helped me during my studies and work. Without your strong support, I couldn't have achieved this level of successful work.

I sincerely express my gratitude to my Professor Timo Hämäläinen for his advice, guidance, and support not only during my thesis work but also during my master's studies. A special thanks to Saman Payvar; you were always available regardless of your own deadlines. Thanks for providing valuable feedback on my thesis.

I want to thank my manager, Petri Uusi-Äijö who trusted in me and supported me during my studies and work. A huge thanks to my colleagues Tero Lehtinen, Tapio Mujunen and Tanja Viiru. Without your wonderful support and help, it would have been extremely difficult.

At the end, I would like to thank my family, who encouraged me with their love, and a big thanks to Reza Allahyarzadeh who was there for me during all the ups and downs of my studies.

Tampere, 1st October 2022

Samaneh Ammari

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABBREVIATIONS

AI	Artificial Intelligence
APU	AI Processing Unit
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BK	Brent-Kung
BN	Batch Normalization
CLA	Carry Look Ahead adder
CLK	clock
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSA	Carry Select Adder
DDR	Double Data Rate
DL	Deep Learning
DLA	Deep Learning Accelerator
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DWC	Depth-Wise Convolution
FC	Fully Connected
FPGA	Field Programmable Gate Array
GAN	Generative Adversarial Networks
GOPS	Giga Operation Per Seconds
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
ILSVRC	Large-Scale Visual Recognition Challenge
IoT	Internet of Things
IP	Intellectual Property

KS	Kogge-stone
LSTM	Long Short-Term Memory
LTU	Linear Threshold Unit
MAC	Multiply and Accumulate
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSB	most significant bit
MSIC	ML-specific Integrated Circuit
NN	Neural Network
NoC	Network on Chip
NPU	Neural Processing Unit
OPS	Operation Per Seconds
PE	Processing Element
PPA	Parallel Prefix Adder
PWC	Point-Wise Convolution
RCA	Ripple Carry Adder
RELU	Rectified Linear Unit
RNN	Recurrent Neural Network
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SMVM	Sparse Matrix Vector Multiplication
SRAM	Static Random Access Memory
TOPS	Tera Operation Per Seconds
TPU	Tensor Processing Unit
UNPU	Unified Neural Processing Unit
VLSI	Very Large Scale Integration
WNS	Worst Negative Slack

1. INTRODUCTION

Recent studies indicate that Machine Learning (ML) algorithms provide considerable benefits, especially in the internet of things (IoT) and edge devices (low-power mobile applications). Edge devices have sensors or actuators that gather data, and instead of transmitting data to the data center for processing and analysis, all the computation could be conducted at the edge. The power and memory capacity of edge devices are limited. Thus, it is challenging to use ML algorithms on edge devices. However, due to data privacy and latency concerns, the demand on using Deep Learning (DL) algorithms on edge devices is increasing.

Several studies have been conducted on designing efficient hardware accelerators. As in ML, especially DL, there is a set of repetitive computations. A collection of specialized hardware is used to do these computations faster and more energy efficient. This set of hardware is generally called a Deep Learning Accelerator (DLA), Neural Processing Unit (NPU), or AI Processing Unit (APU). They are optimized for performing a specific task. There are three aspects to DLA: data reuse ability, fixed computational patterns, and fast memory access. For these reasons, the DL Accelerator designs are moving towards non-von Neuman architectures. It enables the increase of parallelism with the high number of Processing Elements (PEs) utilization.

The goal of this master's thesis is to study and propose the most effective Very Large-Scale Integration (VLSI)-based DL accelerator. The suggested DL accelerator is one of the subsystems in a larger System on Chip (SoC) named Ballast [1]. The proposed DL accelerator will replace the Ballast chip's current DL accelerator. The objective is to introduce a novel DL accelerator that consumes less area footprint and reduces power consumption, while improving performance. Our design strives to be suited for edge devices. The aim of this master's thesis is to investigate the design principles of the DL accelerator and improve every aspect of the design that can be optimized. The focus is on the arithmetic part of the DL subsystem, as it is the bottleneck of the main process. For that reason, different types of multipliers are implemented and compared. The aim is to select the multipliers that are as fast as the critical path of the whole chip. The final hardware architecture should be able to do both standard convolutions and depth-wise separable convolutions.

Figure 1.1 demonstrates a general overview of the Ballast SoC. Besides that, it illustrates the structure of the proposed DL architecture. It consists of Direct Memory Access (DMA), a global buffer, a control unit, a systolic array, and an aggregation core. Other members of the team implement certain components of the DLA. This thesis work focuses on selecting the needed components for the DL accelerator and discussing the reasons behind the selection of different features. Furthermore, it discusses the arithmetic part of DL and how to do the convolution. Finally, it evaluates the different multipliers to determine a fast, energy-efficient, and area-efficient way to do these computations.

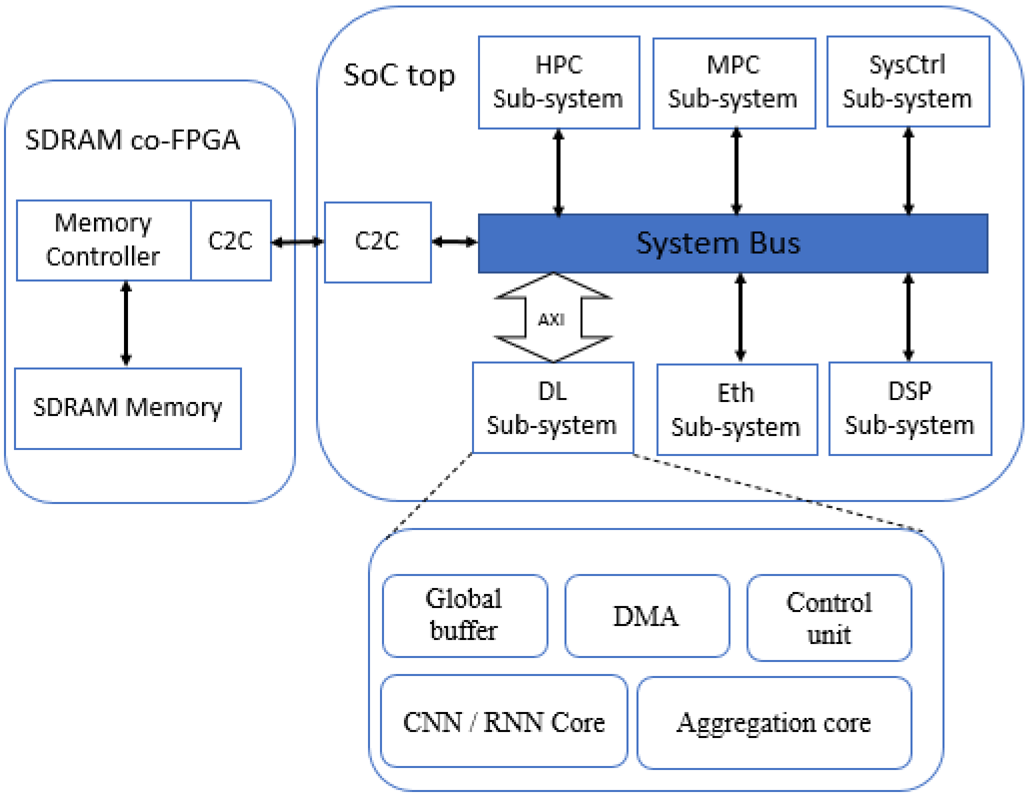


Figure 1.1. *Ballast SoC with proposed deep learning accelerator design*

The thesis work comprises nine chapters. The second Chapter 2 explains the basic concepts that readers with different backgrounds might need. The third Chapter 3 evaluates related works. This thesis consists of two main parts. First, Chapter 4 defines the deep learning design principle, followed by Chapter 5, which proposes a hardware accelerator for edge computing use cases, enabling specific deep learning algorithms. It explains the different parts of the proposed deep learning accelerators and the reason for implementation. Second, Chapter 6 and 7 describe the procedure, techniques, and implementation of different multipliers and adders, respectively. Chapter 8 compares and evaluates different multipliers in terms of the area and power consumption of the DL subsystem. In the last Chapter, 9 the conclusion is discussed.

2. BACKGROUND AND MOTIVATION

This chapter provides a concise explanation of the fundamental principles necessary to comprehend the context. The first segment focuses on machine learning. The second part briefly introduces SoC and embedded systems, followed by a discussion of the significance of the link between ML and SoC in the third section.

2.1 Machine learning

Neural Network (NN) refers to the approach used to design a computer to function similarly to human brains. The name of the NN was inspired by neurons in the human brain. ML is a branch of AI focused on building models and training them with sample data for specific tasks without being explicitly programmed. Supervised and unsupervised learning are the two main types of machine learning. In a nutshell, supervised learning is suitable when the data source is structured, and the model is known. Regression and classification are techniques for supervised learning. Classification can predict a label, whereas regression can predict a quantity.

It is important to specify the following usage cases. The examples of regression would be the agricultural industry for predicting the effect of fertilizers and water on plants, house pricing prediction, and medicine to predict the relationship between blood pressure and drug dosage. In addition, examples of classification applications include the categorization of spam and non-spam emails and the classification of soil and crops. In contrast, unsupervised learning is used for unstructured source of data to discover hidden structures in the data. Products or customer segmentation, and similarity detection are some of the common use cases [2].

Deep learning is one of the most effective machine learning techniques. It uses a neural network to execute complex calculations on big data. Figure 2.1 illustrates the hierarchy between AI, ML, and DL. It is part of ML while it is a subset of AI. Having several layers of neural networks has a significant improvement in ML. There are various types of DL methods, such as Deep Neural Networks (DNNs), Convolutional Neural networks (CNN), Recurrent Neural networks (RNN), Multi-Layer Perceptron (MLP), Linear Threshold Unit (LTU), Generative Adversarial Networks (GAN), Autoencoder, Distributed Learning, Transformer, Deep Feedforward Network, Training Feedforward Network, etc.

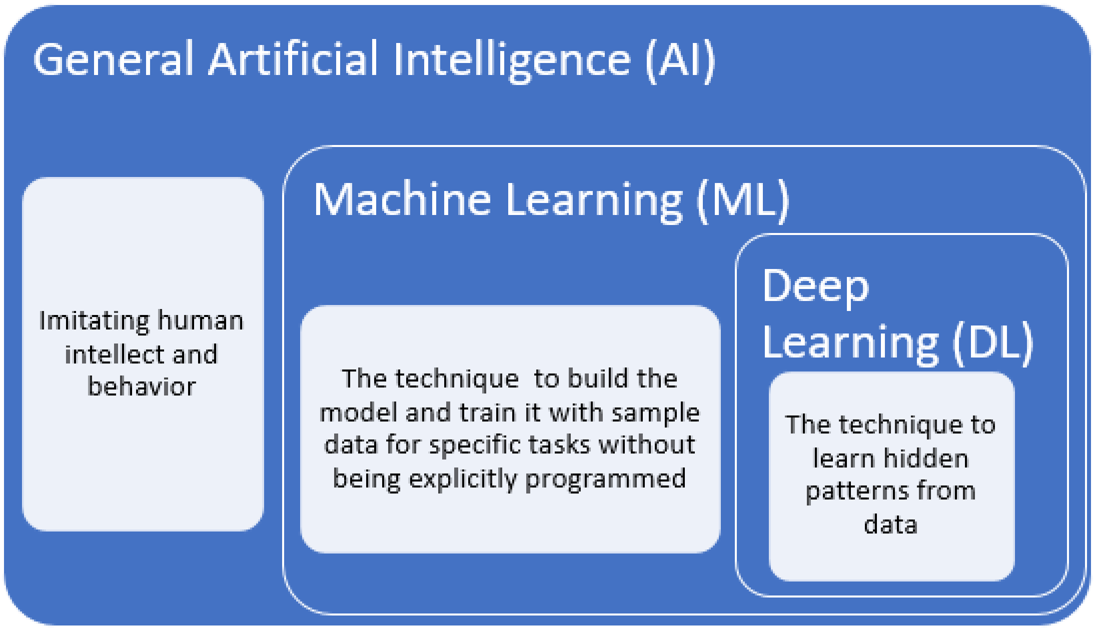

General Artificial Intelligence (AI)
Imitating human intellect and behavior
Machine Learning (ML)
The technique to build the model and train it with sample data for specific tasks without being explicitly programmed
Deep Learning (DL)
The technique to learn hidden patterns from data

Figure 2.1. *Taxonomy of Artificial intelligence (AI)*

Brief descriptions of the three most prevalent neural networks are provided below.

CNN is among the most prominent NN models. It is derived from the mathematical idea of convolution, which is the process of computing the integral of two functions while the shape of one of them is modified and shifted. CNN can extract specific features or patterns. Briefly, in each convolution, the filter slides over the input to detect special patterns such as corners, edges, lines, objects, texture, shape, etc. For instance, a filter that can identify objects is known as an object detector. A CNN comprises of three main layers: an input layer, hidden layers, and an output layer. All layers between an input layer and output layer are called hidden layers. Typically, it also has a Fully Connected (FC) layer and a pooling layer. Each layer contains a set of neurons that have specific weights and biases. The convolution layer applies the filter to an input. As a result, a feature map is produced. If the same filter is applied repeatedly to the input, it will produce the feature map. The ability to learn many filters in parallel is the most important innovation of CNN. LeNet [3], AlexNet [4], VGGNet [5], GoogLeNet [6], ResNet [7], and ZFNet [8] are famous variation of CNN models. These kinds of models are mostly applied to computer vision applications.

Remembering the past states is a unique feature of RNN. These memories are stored in special nodes. Each node gets the data and the previous state as inputs. In addition, the output relies on the input and the previous computation. One of the most well-known RNNs is Long Short-Term Memory (LSTM). This kind of network is efficient when historical characteristics are crucial for making a choice. It dominates predictive

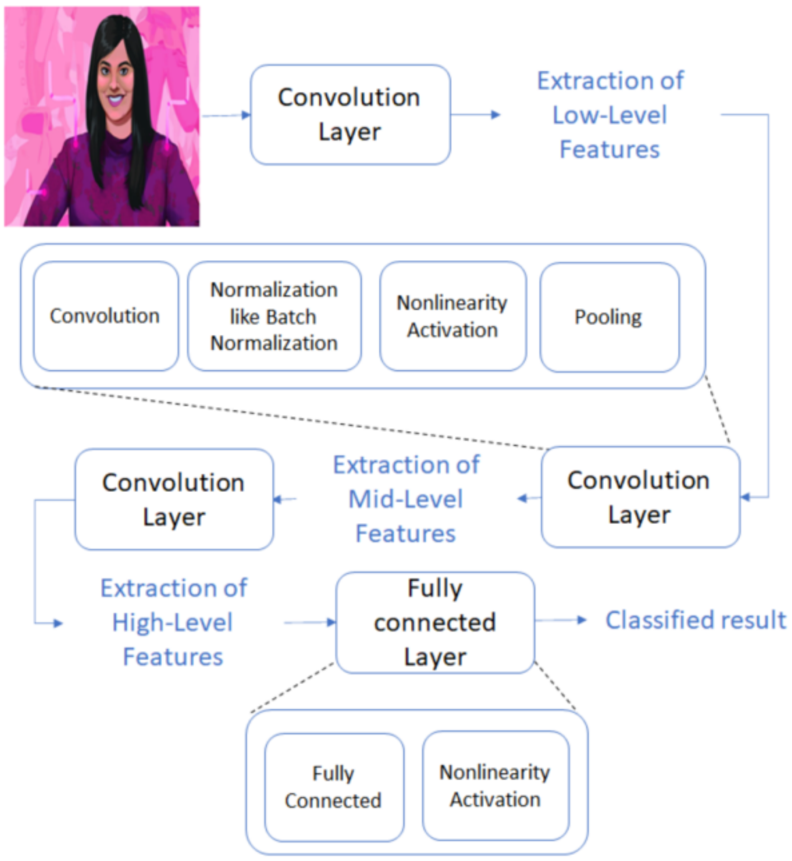


Figure 2.2. *Deep CNN Example of classification application*

applications like as the stock market, speech recognition, and language modeling.

DNNs have more than one layer between input and output. It extracts features from the huge amounts of data available without human interaction. Collective layers are cascaded to extract the feature. The most common NN layers are convolution, FC, activation function, and pooling. Figure 2.2 demonstrates the general process of DL neural networks as an example in image classification. Usually, the input is an image with a dimension of 256*256 pixels, 128*128 pixels, or sometimes a reduced size to save some calculations. The ML model trains with the available datasets to produce weights or kernels. In other words, a filter is created during the training. Filters can be programmed to detect features like straight edges, simple colors, and curves. The kernel convolves over the entire area of input image features to extract the features.

Each of these techniques is suitable for specific applications. Generally, ML utilization shows considerable improvement in computer vision tasks, speech recognition, robotics, self-driving cars, detecting cancer, playing complex games, and so on. NN algorithms may be suitable for more than one application. In these cases, insightful evaluation of NN algorithms determines which of them would be most suitable and provides the highest accuracy for a given application. Some of the applications for different NNs are shown

CNN-DNN	RNN-LSTM
Image segmentation	Trend Forecasting
Image classification	Detecting Anomalies
Medical image analysis	Generating Image Descriptions
Image and video recognition	Language Modelling and Generating Text
Natural Language processing	Speech Recognition
Search engine	Video Tagging

Table 2.1. *Neural Networks (NN) and some of their common use cases*

in the table 2.1 below. The focus of this master's thesis is mostly on supporting the DNN. The main reason is that it can widely support so many applications.

Table 2.2 demonstrates the comparison of the different NN models in terms of the number of parameters and top-1 accuracy based on the ImageNet dataset [9]. The top-1 accuracy rate is a factor for the ground truth against the first predicted class. Another good factor for comparison is the top-5 error rate, which compares the ground truth against the first five predicted classes. According to the table 2.2, the MobileNet model gives high accuracy, despite the low number of parameters. MobileNet is suitable for computer vision tasks and requires less computation.

Model	Number of parameters	Top-1 Acc ImageNet (%)	Year
AlexNet [10]	60M	63.3	2012
VGG16 [5]	138M	74.4	2014
GoogLeNet [6] [11]	4M	68.3	2015
ResNeXt-101(64x4) [12]	83.6M	80.9	2017
Xception [13]	23M	79	2017
MobileNetV3 [14]	5.4M	75.2	2019
InceptionV3 [15]	24M	78.95	2020

Table 2.2. *Summary of popular NN models in ImageNet-1k only*

2.1.1 Convolution

Convolution is the heart of CNN and DNN; It extracts features by convolving the trained filters on input feature maps. There are different types of convolutions, for example, direct convolution, 1D convolution, and depth-wise separable convolution. To illustrate the mathematical complexity of convolution, let's examine the definition of direct convolution. As the equation 2.1 standard convolution demonstrates, there is very much multiplication. The input feature map (i) has height (H), width (W), and the filter (f) is represented with height (R), width (S). The number of input channels as parameters (C) is the same in both input activation and filter. The output feature map (o) has height (P), width (Q), and biases (b), batch size (n) and given stride size (U) are the parameters used to show the convolution. Standard convolution [9]:

$$O_{[p][q][m][n]} = (\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S=0} i[n][c][U_p + r][U_q + s] * f[m][c][r][s]) + b[m] \tag{2.1}$$

$$0 \leq n < N, \quad 0 \leq m < M, \quad 0 \leq p < P, \quad 0 \leq q < Q,$$

$$P = (H - R + U)/U, \quad Q = (W - S + U)/U$$

A simple pseudo code in Listing 2.1 depicts how standard convolution works.

```
for r = [0:R]:
 for s = [0:S]:
  for p = [0:P]:
   for q = [0:Q]:
    for c = [0:C]:
     for m = [0:M]:
      for n = [0:N]:
        output[p][q][m][n] += Weight[r][s][m][c] * Input[p+r][q+s][c][n];
```

Listing 2.1. *Standard convolution*

Corresponding for loops in listing 2.2 illustrate depth-wise separable convolution, which consists of Depth-Wise Convolution (DWC) and Point-Wise Convolution (PWC), in other words, the filtering stage and combination stage. It is the feature of the MobileNet NN [16].

```
for n = [0:N]:
  for c = [0:C]:
    for r = [0:R]:
     for s = [0:S]:
      for p = [0:P]:
        for q = [0:Q]:
          output[p][q][c][n] += Weight[r][s][1][c] * Input[p+r][q+s][c][n];
```

Listing 2.2. *Depth-wise convolution*

```
for p = [0:P]:
  for q = [0:Q]:
    for m = [0:M]:
     for c = [0:C]:
      for n = [0:N]:
        output[p][q][m][n] += Weight[1][1][m][c] * Input[p][q][c][n];
```

Listing 2.3. *Point-wise convolution*

2.1.2 Training and inference

Training and inference are two fundamental concepts in ML. Training indicates the process of producing the NN model. In this process, a DNN learns how to analyze a set of data and make decisions. Inference, on the other hand, applies the generated ML model to novel data to anticipate the outcomes [17].

It is noteworthy to mention some well-known datasets and DNN algorithms. Usually, these datasets are used to train a NN model. The three most available datasets are MNIST, CIFAR-10, and ImageNet. Their complexities increase in the same order. ALexNet [4], GoogLeNet [6], ResNet [7], are popular DNNs that won the Large-Scale Visual Recognition Challenge (ILSVRC) in image classification in 2012, 2014, and 2015, respectively.

There are some open source ML frameworks for training and inferences, such as TensorFlow [18], Pytorch [19] , Keras [20], Caffe [21] and Microsoft Cognitive toolkit. These mentioned frameworks are used to train the NN Model.

2.2 System on Chip and Embedded systems

Embedded systems play an essential role in our lives. There is a good chance we are already using multiple embedded systems in our daily lives, and they can be found in cars, home appliances, digital watches, etc. In general, an embedded system is a term describing a computer system combining both hardware and software designed to perform a dedicated function, which can be part of a larger system, hence embedded in it. This system can utilize multiple embedded processors, such as a System on Chip (SoC), Field Programmable Gate Array (FPGA), microcontrollers, and some electronic components.

An SoC is an integrated circuit (IC) that combines many components of an electronic system into one chip, such as the Central Processing Unit (CPU), Graphics Processing Units (GPU), memory interfaces, input/output interfaces, and may contain digital, analog, or mixed signals and often signal processing units. An SoC could be designed for special use cases or functions such as fast data processing or edge computing and may include special accelerator units such as Tensor processing units (TPU) [18].

An Application Specific Integrated Circuits (ASIC) and a FPGA are the ideal choices when designing hardware for a particular task. The significant advantage of them is customization for specific needs. In contrast to ASIC, FPGA allows clients or users to change designs after manufacturing.

However, Custom ASIC has many advantages to mention compared to the CPU, GPU, FPGA, and other types of computing systems. ASICs, on the other hand, have a few drawbacks, including a long time to market due to the lengthy design, research, and development process, whereas FPGA is appropriate when time to market is critical. However, the power consumption of FPGA products is much higher than ASICs, especially on edge devices and mass-produced IoT. Table 2.3 compares the characteristics of different kinds of computing systems.

Characteristic	**CPU**	**GPU**	**ASIC**	**FPGA**
Development cost	Low	Low	High	Medium
Unit cost	High	High	Low	High
Time to Market	Short	Medium	Long	Medium
Flexibility	High	Medium	Low	Very High
Power consumption	High	Very High	Low	Medium

Table 2.3. *Comparison of various chip platforms*

It is essential to remember that the appropriate hardware solution is chosen in accordance with the application, time to market, and design criteria. It is possible to categorize most DL accelerator designs between general-purpose architectures like GPUs and CPUs and neuromorphic architectures like [22].

With the development of technology, the need for smart, portable gadgets increases. In edge devices, local memory processing and power consumption are bottlenecks. Hence, there is a need to develop either specific hardware for each unique task or flexible hardware. Thus, it is possible to have optimal hardware that can perform the same assignment much faster. It is recommended to design specific hardware to get the best performance and accelerate the speed of computation.

2.3 Relationship between System on Chip and Machine Learning

With the most recent breakthroughs in information technology, ML has found opportunities to expand in several technologies. Nowadays, ML is being applied extensively in industrial and edge processing, such as speech recognition, computer vision, image classification, object detection, and many other use cases. There are numerous ML algorithms, like state-of-the-art DNNs. Even though DNNs predict the results with high accuracy, they introduce high computational complexity. It is possible to deploy ML algorithms on different hardware platforms; the cloud, GPUs, CPUs, FPGAs, and ASICs are the most famous hardware platforms.

SoCs and ML can become related in two ways. Either ML is used to design an SoC or design an SoC for a specific ML application. The emphasis here is on the latter. There is a tradeoff among various hardware platforms. The platform is selected based on the application requirements.

However, the tendency to process data locally is increasing to protect data privacy, especially in edge applications. In some use cases to prevent fraudulent activities, hacking, phishing, and identity theft, the cloud is not a viable option. On the other hand, CPU-based platforms are insufficient for parallel computing, which DNNs need. They support several instructions sets, and multiple access to external memory per instruction has a negative effect on power consumption, although there are only a few repetitive instruction sets needed for specific tasks.

GPUs are commonly utilized for accelerating ML algorithms. GPU-based platforms are much better than CPUs at deploying DL. GPU's multiple cores provide astonishingly fast, highly parallel computations. GPU-based platforms can have high throughput, but they use too much power to be useful for edge devices. [23] has compared FPGAs against other hardware platforms. According to their research, FPGAs and ASICs surpass GPUs and CPUs in terms of energy efficiency and performance. Moreover, ASICs rise above all of them.

[24] compares hardware options for neural network acceleration in terms of power efficiency and the number of processing units. According to the book, ASICs and FPGAs with customized compute logic have higher power efficiency than GPUs with single instruction, multiple data executed for general-purpose computing. And the CPU with a few cores for mobile/edge has the lowest power efficiency.

The DNN algorithms on GPUs with the Von-Neumann architecture are computationally more costly than the ASIC's design. Nowadays, the trend is to use the ASIC platform for ML, especially for DNNs. As the number of weight parameters is high in the DNN algorithms, it demands data movement between the processing unit and memory, which is expensive due to latency and energy consumption caused by the memory wall. DLA's design is shifting toward Non-von Neumann architecture to support parallelism. As a result, it enables fixed memory access and computational patterns with deterministic data reuse-ability, and it is easy to increase parallelism with more PE utilization.

General purpose processors such as CPU and GPU have Von Neumann architecture like figure a in 2.3. These kinds of processors consume a considerable amount of power, and they are not suitable for low-power edge devices. On the other side, by implementing the non-von Neumann architecture or even computing in memory, it is possible to carry out higher performance, energy efficiency, and cost-effective platforms. As a result, DL accelerators designed by FPGAs or ASICs follow a semi-spatial structure like in figure b 2.3. Therefore, as ML develops quickly, hardware architecture should be designed to support ML computation. There is a need for a hardware architecture to execute the highly computational neural network algorithms fast. The spatial hardware architecture helps to increase the speed of computations. This can happen by bringing memory near the computation unit or inside it. Thus, there will be less latency for transferring the data

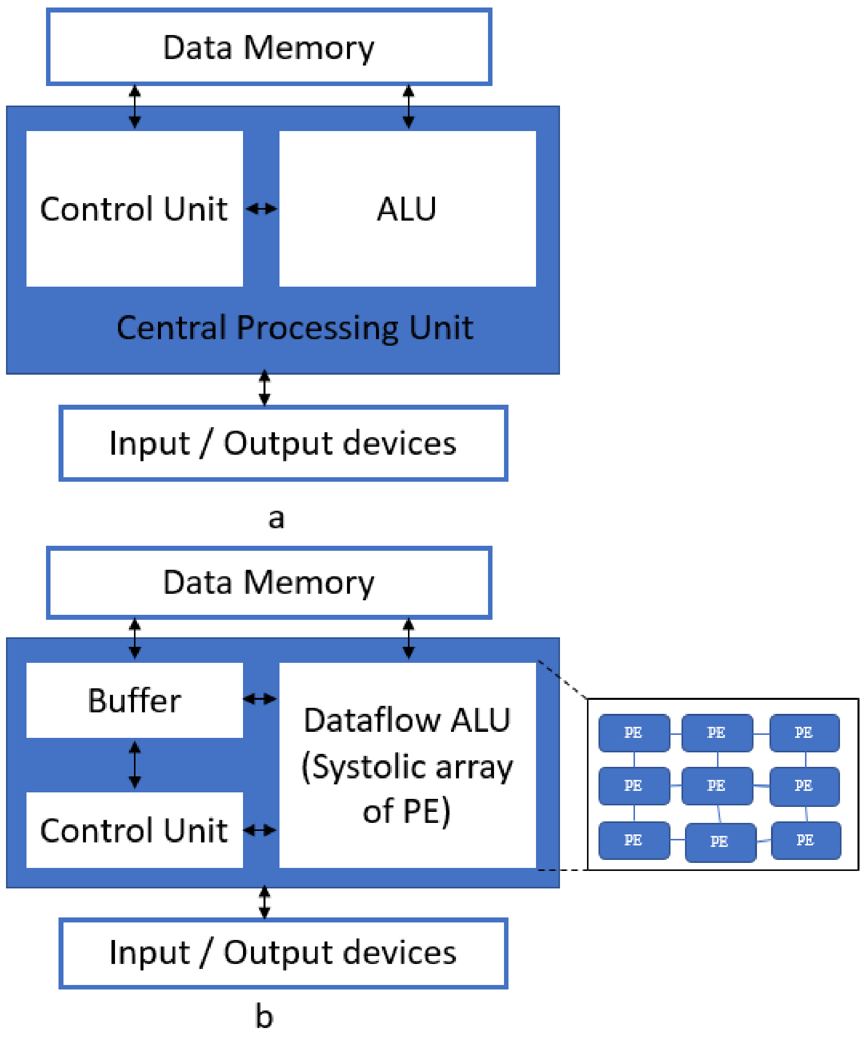


Figure 2.3. *Two common platform structures for ML: a) General purpose processors b) Common spatial architecture*

between different units.

ML is all about computation, and it consumes considerable amounts of power. But the power source is limited to edge devices. Thus, it is challenging to design an ML accelerator that consumes a smaller amount of energy. According to [25], Moore's law is going toward the end. Power becomes a key factor. Therefore, to have a faster processor instead of a general-purpose processor, build a more heterogeneous architecture that only does one specific task. The software-centric, hardware-centric, and combination approaches are the ones suggested. So, there is a need to employ domain-specific hardware in a computing system and use the hardware and software co-design to boost performance. In other words, to do co-design, the proposed DL accelerator is designed for a specific NN algorithm. In this thesis, the MobileNet algorithm [16] is selected as it can cover many applications of ML.

3. RELATED WORK

3.1 Deep Learning Accelerators

In recent years, DNNs and other domain-specific ML accelerators have gained a lot of attention in the computer architecture community. The amount of literature available on DL accelerators is growing quickly. Table 3.1 demonstrates several deep learning accelerators and compares them based on their technology, chip area, power consumption, and overall performance.

Platform	Technology (nm)	Area (mm^2)	Power (mW)	Performance (GOPS)
Ballast [1]	22	2,63	439,3	80.80
Eyeriss V2 [26]	65	≈30[1]	-	193.7
CENNA [28]	65	1.38	47.34	86
EIE [29]	45	40.8	590	102.4
VWA [30]	40	1.56	154.98	146.33
DaDianNao [31]	28	67.7329	15970	5580
ParaML [32]	65	3.51	596	1056
Bert-Marian [33]	40	2.4	76	102

Table 3.1. *The comparison of DNN accelerators*

Ballast [1] has an AI-subsystem based on NVDLA [34]. It has 32x8 bit MAC units and operates at a 750 MHz clock frequency. Eyeriss version 2 [26] reports the throughput and energy efficiency of their designed DL accelerator with four different DNN algorithms. The values in Table 3.1 are for the MobileNet neural network inferences on their DL accelerator. CENNA [28] is the 16-bit CNN accelerator which uses matrix multiplication for

[1] Eyeriss v1 area is 1394k NAND-2 gate reported in [26] which is also reported 16 um^2 in [27]. Therefore, Eyeriss v2 area which is reported 2695k NAND-2 gate in [26] is estimated $\approx$ 30 um^2.

doing the convolution. As shown in Table 3.1, it produces good results.

Chen et al. [35] focus on different approaches to reduce data movements such as Weight stationery, Output Stationery, No local Reuse (NLR), Row Stationary (RS). Then claim that the RS is the best in power and energy consumption. Calculating the 1D Row Convolution in PE is named row stationary.

State-of-the-art, EIE [29] is an efficient inference engine that supports sparsity. It utilized a dedicated accelerator, performing Sparse Matrix Vector Multiplication (SMVM) and handling weight sharing. EIE saves a considerable amount of energy with four actions: moving the data from Dynamic Random Access Memory (DRAM) to Static Random Access Memory (SRAM), then using sparsity, sharing weights, and finally, skipping zeros caused by activation. Another AI accelerator that supports sparsity is NullHop [36]. The authors claim that an efficiency of 368 percent is possible because of its zero-skipping pipeline and high MAC utilization.

VWA [30] is a hardware-efficient CNN accelerator that can implement different convolution filters with good results. It combines systolic array architecture with vector-wise input and weight data to preserve regular structure. As a result, they manage to reduce expenses while still satisfying the aforementioned requirements for flexibility.

Another well-known DLA is DaDianNao [31]. It stores all the computation-related data in local memory to reduce the number of memory accesses. The author claims accomplishments in both high performance and low power consumption.

ParaML [32] or Polyvalent Multi-core Accelerator for ML is an architecture that supports ten different ML techniques (k-NN, k-means, linear regression, SVM, DNN, naive bayes, classification tree, LVQ, parzen window, PCA) and is a flexible design that can accommodate for increased data size and is energy efficient for different ML scenarios. Bert-Marian [33] is a low-power precision-scalable processor for CNN which has 256 parallel PU and runs at 204MHz, it uses the sparsity of convolutions and implements dynamic scalability, it utilizes 16x16 MAC array.

Table 3.2 shows all these different accelerators sorted by their performance per power consumption (GOPS/W). Because these are implemented with different technologies ranging from 22 to 65nm, comparing the area or power is not a one-to-one comparison, so sorting them by their respective performance per power consumption is a better way to demonstrate their efficiency. Figure 3.1 visualizes the results, and as some implementations have power consumption on a scale of thousands more than others, the X axis is cropped at 600 to be able to view the smaller values. CENNA [28] has the best GOPS/W result, followed by ParaML [32], even though both are implemented in 65nm technology. Although DaDianNao [31] had the highest performance, we can see that it has very medium GOPS/W compared to other accelerators. As there is no other 22nm accelerator, we can't compare Ballast in more detail with others correctly.

Platform	GOPS/W
CENNA [28]	1816,65
ParaML [32]	1771,81
Bert-Marian [33]	1342,11
VWA [30]	944,186
DaDianNao [31]	349,405
Eyeriss V2 [26]	243,036
Ballast [1]	183,929
EIE [29]	173,559

Table 3.2. *Sorting Table 3.1 by Performance per power consumption*

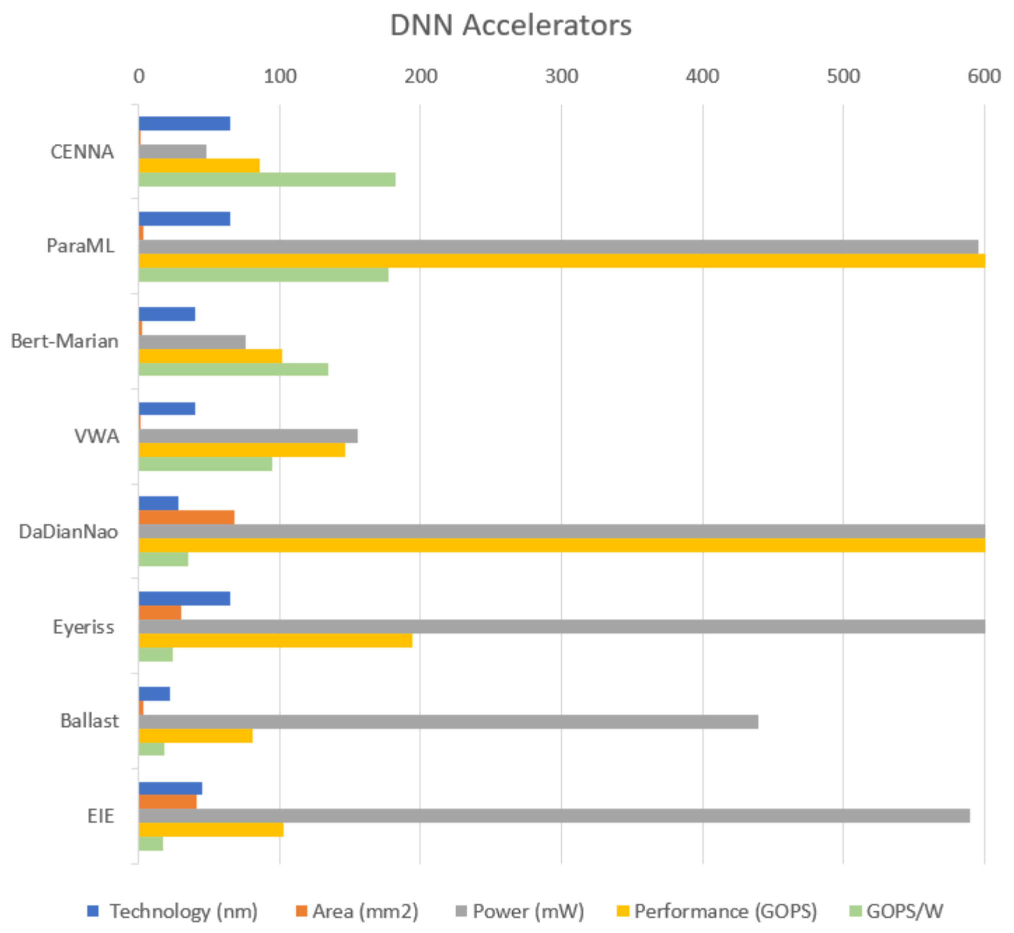


Figure 3.1. *Visual comparison of Table 3.1*

Chen et al. [26] explore sparsity and architectural design for sparse DNN models. By using Rectified Linear Unit (Relu), the decoder layer and pruning sparsity happen to DNNs. It is possible to use this sparsity to improve energy efficiency and the speed of processing. These can happen in two ways: either skipping, gating the MAC computation, or compressing the weights and activations. The latter helps to decrease data movement and storage. However, it is not as easy as it sounds. There are two main challenges: first, access pattern irregularity, and second, a different PE has an uneven workload. It is noteworthy to mention that Eyeriss V2, in contrast with other versions, supports sparsity. Eyeriss v1 [27] does not support sparsity and it has 1394k NAND-2 gate area, while Eyeriss v2 [by supporting sparsity] consumes 2695k NAND-2 gate area, approximately two times more. It vividly shows that supporting sparsity increases area. Sparsity is not the only reason for the increase in area; another reason is the use of Single Instruction, Multiple Data (SIMD), which doubles the MAC operations.

Bing et al. [37] is the FPGA based DL accelerator, which supports depth-wise separable convolution. It consists of depth-wise convolution and point-wise convolution. MobileNet and ShuffleNet are examples of ML models which use depth-wise separable convolution. The MobileNet architecture introduces depth-wise separable convolution layers [16]. In this architecture, 2D convolution is replaced by 1D convolution along with depth-wise convolution. Reducing the number of parameters as well as calculations are the reasons for using this kind of convolution, while keeping the loss of precision limited.

Table 3.3 demonstrates three convolution-recurrent neural networks (CNN-RNN), and compares them in terms of area, power, and throughput. Because of the space and power requirements, there are few designs for the combined CNN-RNN accelerator. Shin et al [38], [39] propose a heterogeneous ML-specific integrated circuits (MSICs) architecture called DNPU. There is a top controller, the MLP-RNNs processor, and a CNNs processor inside the DNPU in order to support gesture and action recognition as well as image captioning. As a consequence of supporting CNN and RNN at the same time, this accelerator consumes more areas. Furthermore, the Unified Neural Processing Unit (UNPU) [40] accelerator employs CNN-RNN cores. It is designed specifically for dialogue generation and emotion recognition tasks. An interesting point about UNPU is that weight and input features are concatenated into a 1D vector.

CNN-RNN	Technology (nm)	Area (mm^2)	Power (mW)	Performance (GOPS)
UNPU [40]	65	16	279	691.2
YIN [41]	65	19,36	447	409.6
Zeng [42]	FPGA-ZU5EG	N/A	8000	690.76

Table 3.3. *Related works with CNN-RNN architecture at 200 MHz frequency.*

3.2 Multipliers

Convolution is calculated by multiplying and accumulating input feature maps and weights. Billions of MAC operations are required, which results in intensive data movements. Table 3.4 shows the number of MACs for different DL algorithms.

In most DLA designs, the terms MAC and PE are mentioned a lot. A PE can contain MAC units, a small memory buffer, and a control unit. For example, [43] uses four RISC-V processors as PEs, and each RISC-V core has a buffer, controller, and MAC. Consequently, these terms are sometimes used instead of each other. When PE is mentioned, the focus is on the number of MAC units it contains.

Deep Learning Algorithms	Num of PE/MAC units	Num Weights
MobileNet [16]	569 M	13 M
Inception [44]	5.74 B	52 M
AlexNet [4]	724 M	61 M
VGG16 [5]	15.5 G	138 M
ResNet-50 [7]	3.9 G	25.5 M
GoogLeNet [6]	1.43 G	7 M

Table 3.4. *Computation requirements MACs and weights of various deep neural networks*

Table 3.5 shows the number of MAC/PE units of various deep learning accelerators. According to [26] MAC units consume around 65 percent chip area of their work, which is the similar situation in most of the DL accelerator designs. Moreover, it indicates that the number of MACs that contain multipliers is high in each DLA. Thus, it is important to have a multiplier that consumes less chip area and power. For example, the systolic array in Eyeriss [27] architecture consists of 168 MAC units. It's interesting to note that in Eyeriss V2 [26], the bit width has been reduced from 16 to 8 bits. The array of MAC unit structures, in particular multipliers, will occupy a vast chip area and consume a considerable amount of power.

Camus et al [45] evaluate the implementation of MAC units in various DL accelerators. Using the precision-scalable MAC architecture is one of the common ways, such as the Deep Neural Processing Unit (DNPU) [46], Unified Deep Neural Network (UNPU) accelerator [40].

Platform	Num of PE/MAC units	Area (mm^2)	Bit Width
Ballast [1]	256	2.63	8-bit
Eyeriss v1 [27]	168	16	16-bit
Eyeriss v2 [26]	192	$\approx$ 30	8-bit
Bert-Marian [33]	256	2.4	1-16 bits
EIE [29]	64	63.8	4-bit
VWA [30]	168	1.56	16-bit

Table 3.5. *Comparison in the number of MAC/PE, area; ASIC platform.*

4. DESIGN PRINCIPLES OF DEEP LEARNING ACCELERATOR ARCHITECTURES

This chapter discusses some principles of designing a DLA. Figure 2.3 demonstrates how DLAs adhere to spatial architecture. It enables the increase of parallelism with the high number of PEs utilized, because there are fixed computational patterns, memory access, and data reusability in them.

4.1 Data Movement

Training and inference of each ML algorithm require abundant amount of data. Only considering the inference, each layer of ML needs a tremendous amount of data transfer across distinct blocks. The transmission of data can become a bottleneck in execution. Data movement and memory data structures influence overall performance, such as energy consumption and throughput, especially in memory-bound systems. According to [47], transferring the data from DRAM (external memory) to ALU consumes 200 times more energy than transferring the data from the register file (RF) to ALU, and data movement between the global buffer (internal memory) and ALU spends six times more energy.

One of the ways to reduce data movement overhead is the use of Direct Memory Access (DMA). While the main processing unit is occupied by other tasks such as reading from external inputs and generating control signals, another part of the system called DMA can enable certain hardware subsystems to access main system memory independently of the CPU. DMA makes the design faster by reducing the time required for data movement. However, even after utilizing the DMA, data movements in the DL accelerator are noticeable, time-consuming operations.

It is critical to remember that enabling sparsity minimizes data movement. Because the sparse weight is no longer in memory, and when the input data is 0, MAC operations are omitted.

4.1.1 Data flow of the systolic array

Going toward the spatial architecture enables data reuse, which reduces data movements. It is noteworthy to highlight that it is necessary to exploit spatial reuse. Otherwise, it leads to the underutilization of parallelism. Multiple MAC units align together to form a systolic array. Hence, since convolution doesn't have ordering constraints, alternative data flows can be implemented. Usually, the connection of MAC units follows one of the data flow structures listed below. The weight stationary is the most popular among others. Notably, there are methods for transferring data across MACs and reducing data movements. The most well-known ones are mentioned by the listing. To understand each item in the for loop, consider I[w] to be the input activations, F[s] to be the filter weights, and O[q] to be the output activations.

- Weight stationary like NVDLA [48], TPU [49], Origami [50] and UltraTrail [51]. Weights are stationary and are obtained once from memory. The connection in the pseudo code in section 2.1.1 has this kind of connection. It can also be represented as:

```
for s = [0:s]:
 for q = [0:q]:
    w = q + s
    O[q] += I[w] * F[s];
```

Listing 4.1. *Weight stationary*

- Output stationary like ShiDianNao [52], DaDianNao [31]. In this kind of data flow, intermediate results known as partial sum are stationary.

```
for q = [0:q]:
  for s = [0:s]:
    w = q + s
    O[q] += I[w] * F[s];
```

Listing 4.2. *Output stationary*

- Input stationary like SCNN [53], This kind of data flow to reduce the data movement, keeps the input feature maps stationary.

```
for w = [0:w]:
  for s = [0:s]:
    q = w - s
    O[q] += I[w] * F[s];
```

Listing 4.3. *Input stationary*

- No local reuse like DianNao [54], as it is obvious from its name there is no data reuse in any local register.
- Row stationary 1D Row Convolution in PE) like Eyeriss Version 1 and version 2 [27]. In this data flow, a row of a kernel and a row of one channel of the input fea-

ture map are stationary.

Based on the findings of [27], the row stationary is much more efficient than the other mentioned data flow. The comparison was between different data flows of the same design of 256 PEs and a batch size of 16 based on the ALexNet convolutional layer.

4.1.2 Dimension of the systolic array

With regards to [55] a roofline model tells how to improve the performance. It is used to figure out the appropriate number of MAC units and the dimension of the systolic array. According to the roofline model, the number of MACs/PEs cannot exceed a certain amount since having an unnecessary number of MACs/PEs wastes resources. Hence, finding the suitable dimension of the systolic array is the most challenging part. A suitable number of MAC units should be chosen so that the DL accelerator performs optimally.

Figure 4.1 depicts the factors influence the number of PEs/MACs. There are seven steps to determining an adequate number of MAC units [56]. These seven steps are the layer shape and size, data flow; predefining the number of PEs/MACs to check the impact of the architecture, dimension of the systolic array, memory capacity by considering the global buffer size, data bandwidth, and various data access patterns.

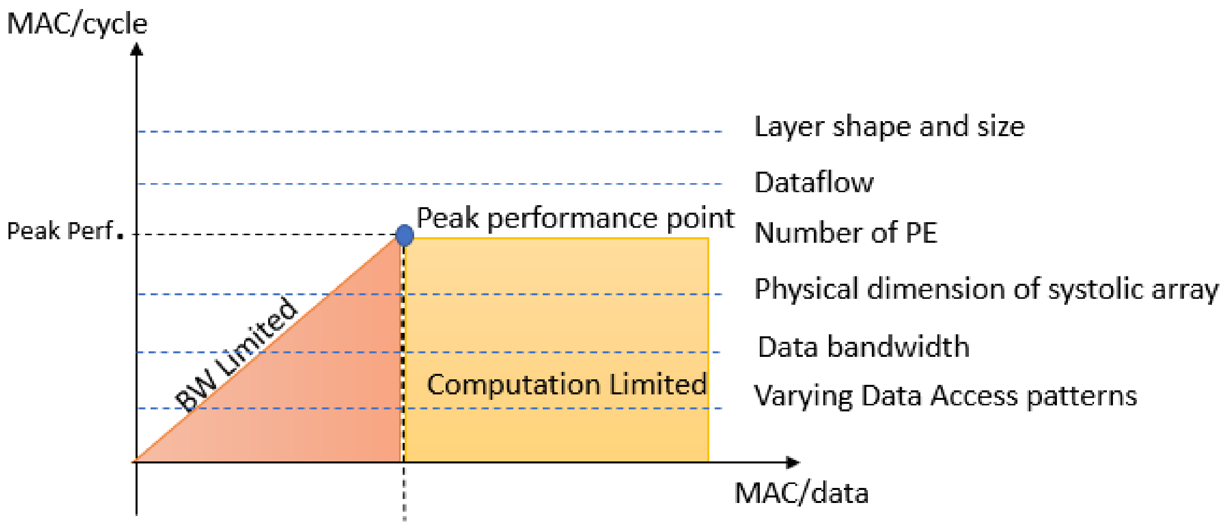


Figure 4.1. *Roofline Model for selecting the number of MAC operation [55]*

4.2 The network on chip of the systolic array

As explained in 4.1.1, a Network on Chip (NoC) should have three features to support data flow: processing with high parallelism by efficiently delivering data between storage and data path; Moreover, it should exploit data reuse to minimize the bandwidth requirement and improve energy efficiency. Finally, it is scalable at a reasonable implementation cost.

Figure 4.2 illustrates four NoC structures suitable for DL accelerators. Unicast networks are represented by a, 1D Multicast networks by b, 1D Systolic networks by c, and broadcast networks by d. Each NoC implementation has pros and cons. As an example, Unicast Network has low reuse and high bandwidth, while broadcast Network is vice versa.

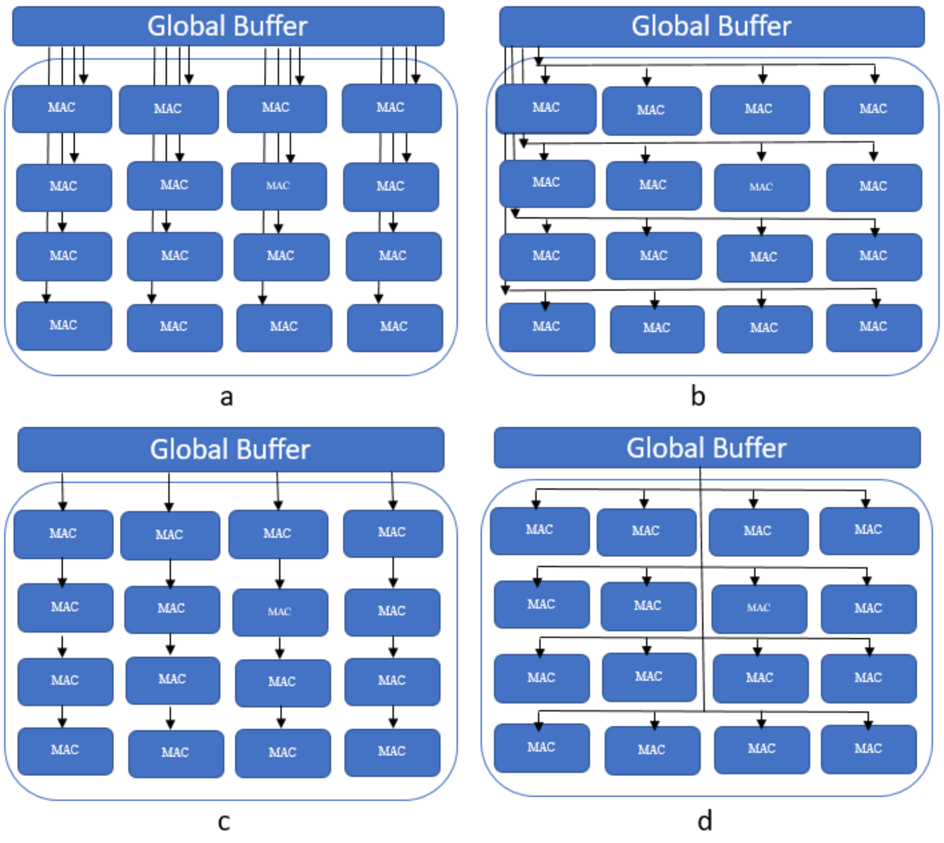


Figure 4.2. *Common Network on Chip*

It must be pointed out how the results data flow would be. According to [57] the results of each MAC unit can either sum apart or sum together. [57] proposed two MAC engines, with different implementation of sum separate and sum together suitable for DNN inferences. Sum separate means that the intermediate results from each PE are apart from each other while sum together means the results form one single output.

4.3 Post processing

Post-processing is the final step of the computational pipeline. Depending on the layer, it may include pooling, activation, Batch Normalization (BN), etc. Here, briefly, some of them are discussed.

There are two types of nonlinear activations: conventional and contemporary. Classical nonlinear activations include the sigmoid and hyperbolic tangent, while current nonlinear activations include the Rectified Linear Unit (Relu), Leaky Relu, and Parametric Relu. The three most popular activation functions are Sigmoid, Tanh, and Relu. The first two are well-known for their mathematical analysis, while the third is well-known for their simplicity. These functions improve the DNN's linearity.

Pooling or more generic downsampling helps to resize the feature map. It is usually executed after the activation. The ability to extract aspects of a picture, such as sharp and smooth characteristics, is the ability of pooling. Maximum, Minimum, and Average are different pooling operations. On the other hand, there is also unpooling/upsampling which increases the resolution of the feature map.

BN is executed between convolution and a fully connected layer or nonlinear activation. In the design of CNNs, batch normalization has become the norm.

$$\text{Equation of the batch normalization:} \quad y = \gamma * \frac{x - \mu}{\sqrt{\delta^2 + \epsilon}} + \beta \tag{4.1}$$

γ : Scale value, β : Shift value, ϵ : Constant, δ : Diviation, μ : Mean, x : input

Figure 4.3 illustrates some of the layers. Convolution extract features by convolving the trained filters on input feature maps. In the convolution, when the the filters are the same size of the input feature map is a special situation, called fully connected layer.

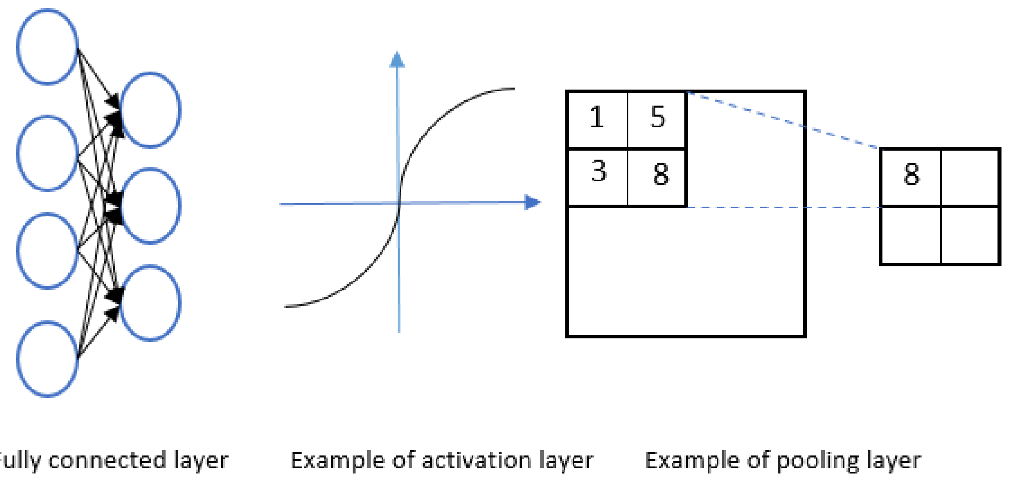


Figure 4.3. *Different layers of Neural Network*

5. PROPOSED DEEP LEARNING ACCELERATOR ARCHITECTURE

This chapter briefly explains the elements of the proposed deep learning accelerator and the reasoning behind them. Figure 5.1 depicts the structure of the proposed DL accelerator. It has five important Intellectual Properties (IPs): the Global Buffer, the DMA, the Control Unit, the CNN/RNN core, and the Aggregation Core.

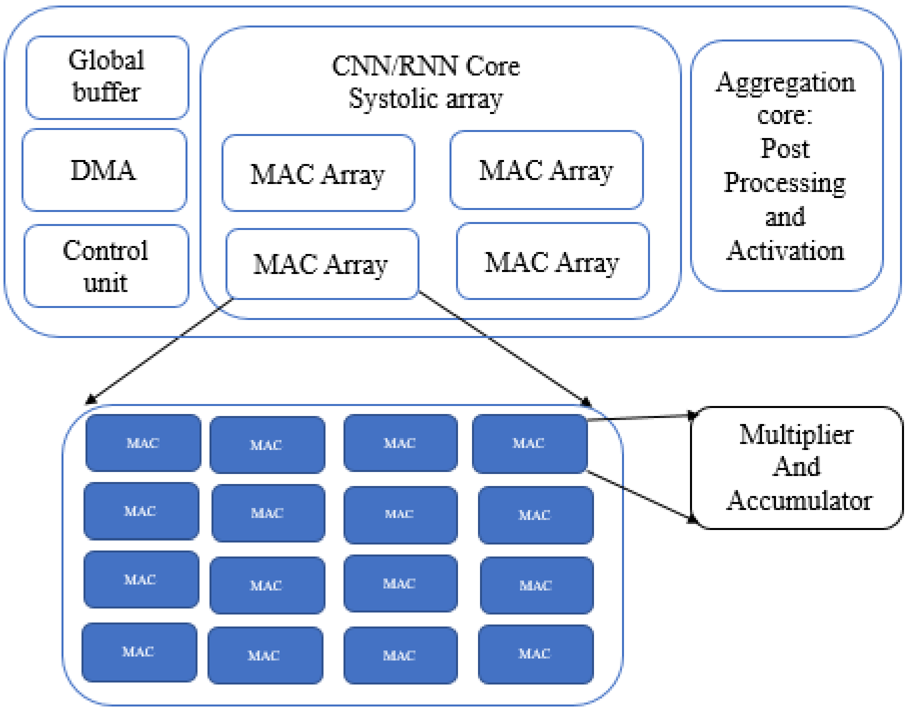

Global buffer
DMA
Control unit
CNN/RNN Core Systolic array
MAC Array
MAC Array
MAC Array
MAC Array
Aggregation core: Post Processing and Activation
MAC
Multiplier And Accumulator

Figure 5.1. *Top level overview of proposed the DL accelerator*

5.1 CNN/RNN Core

To keep the design highly flexible, the architecture should be configurable so that it can perform on both CNN and RNN. This subsystem is constructed of four MAC arrays.

Each MAC array consists of 64 MAC units which in total there 4*64 = 256 MAC units. Figure 5.1 only illustrates 16 of them. In addition, each MAC unit has a multiplier and an adder.

The number if MAC units is selected based on the roofline model 4.1.2. The first constraints would be the layer size and shape. This affects the maximum workload parallelism, which is based on the selected NN model. The second step, the dataflow loop nest will limit the maximum data flow parallelism. For example, here we would like to have weight stationary dataflow, this means that each MAC stores one weight. Then in the third step, the finite number of MACs is defined, based on theoretical peak performance and evaluating different scenarios we conclude that based on our use cases if 512 MAC units is selected we will end up with multiple idle MACs at the processing time which reduces the overall performance therefore reducing that to 256 we reach our peak performance, 128 MAC units result in under performance and increased latency, this constraint the number of maximum MAC parallelism to 256 units. The dimension of the MAC array and fixed storage capacity for intermediate data will bound the number of active MAC units.

5.1.1 MAC Unit: Multiplier and Adder

The MAC unit is the smallest unit in the whole DL submodule. There are different ways to implement MAC units. Here, a simplistic MAC unit is considered. It consists of an adder and a multiplier. Figure 5.2 demonstrates the inputs and outputs of each MAC unit as well as the relationship between multipliers and adders. For building the MAC unit different multiplier and adder algorithms are implemented and compared. Chapter 7 explains more about it. Here is a brief primer on multipliers and adders.

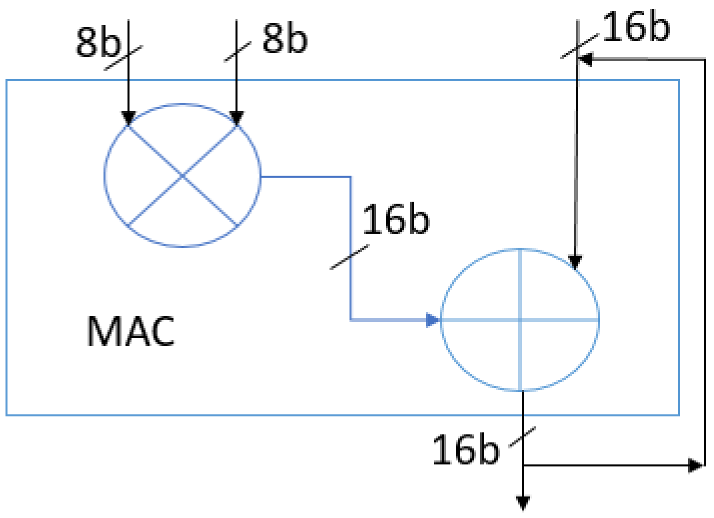


Figure 5.2. *MAC unit structure*

As illustrated in section 2.1.1 multiplication and addition operations are at the heart of convolution. The multiplier is one of the critical parts of the DL subsystem since it's both repeated multiple times and consumes a tremendous amount of chip area and power. Therefore, designing an efficient multiplier has a great impact on the overall performance of the subsystem. On the other hand, the bandwidth that the multiplier can handle will affect the on-chip buffer size. For these two main reasons, it is valuable to implement the multiplier with logic gates that can calculate 2-bit, 4-bit, and 8-bit values. Similarly, addition is repeated several times. Although they do not consume too much area, the multipliers and adders' goal are to perform convolution. There are several different ways to do the convolution, such as Toeplitz, Gauss's complex multiplication transform, and Strassen's Matrix Multiplication Transform [58], [59].

Another notable factor is that the bit width of the outputs of multiplication is twice the bit width of the input. For example, the output of 8-bit multiplication is 16 bits. As a result, it should be truncated to 8 bits consequently some values will be lost. This affects the accuracy of the ML result. The truncation should occur before being written back to the memory.

This DL accelerator also supports the characteristic of sparsity. This feature minimizes MAC operations by bypassing the MAC units when the input data is zero. The sparse weight will no longer be saved in the memory. However, additional zero values may be formed during some process like Relu activation. Therefore, it is feasible to filter out these needless MAC operations by placing one flip flop before the MAC units and skipping MAC operation if the input is zero.

5.1.2 Data Movement

Another way to reduce the data movement is to decrease the bit width of the weights and input features. Hence, in the proposed design, input features and weight data are packed in any 2, 4, or 8-bit format. The designed equivalent multiplier can perform the 2,4,8-bit computation. As a result, there is no need for data manipulation inside the DL accelerator. The [60] is an excellent example of limiting weights and activation to 4 and 8 bits, respectively. It is important to highlight that reducing the precision will decrease memory bandwidth, storage costs, energy per MAC operation, and memory access. However, it will have an impact on accuracy. Then, it is critical to be careful to maintain accuracy.

The proposed DL accelerator supports depth-wise separable and normal convolution. Thus, the connections between the MAC units are important. They should be connected in such a way that both computations are feasible with one systolic array. The key rationale is that this method of operation will save area while increasing efficiency. Furthermore, because we will be doing the calculation layer by layer in each NN model, there will be no requirement for concurrent computation of normal and depth-wise convolution.

There is a significant challenge in selecting the suitable NoC due to various layers and models. Each model requires a different amount of data reuse. Considering the MobileNet model, the DL accelerator's proposed NoC would be a row-stationary data flow with a broadcast NoC.

5.2 DMA

To decrease the data movement, data is read once from the external memory and stored in the internal memory. The data is then distributed across other units, particularly systolic arrays, via DMA. Figure 5.3 depicts the data path of deep learning accelerators. A control bus links the top controller with other IPs. And between DMA and other IPs, there is a data bus.

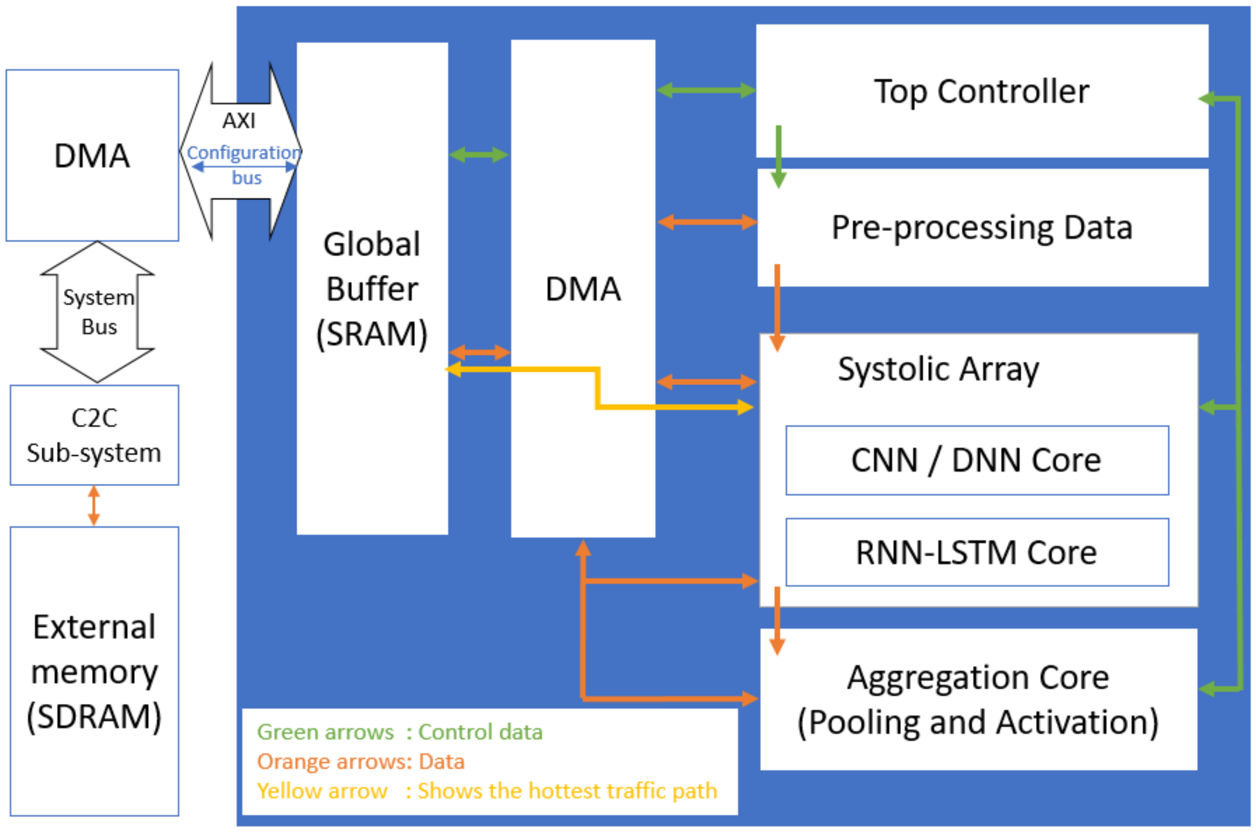


Figure 5.3. *Data movements between major blocks*

In hardware design, it is always important to specify the input and output of intellectual property (IP). In the proposed deep learning accelerator, data transfer occurs through the AXI protocol and DMA. DMA has a configurable interface to transfer the data. It maximizes the amount of data that can be transferred between external and internal memory. One of the critical roles of a DMA is to remove the requirement for a processing unit to transfer data. It should sync with other relevant components to be utilized effectively.

Interrupt signals are sufficient for initiating the data transfer.

5.3 Control Unit

The control unit is a component that oversees the operation of the accelerator. It interacts with different parts of the design through control signals and interconnects [61]. The resources are managed by the control unit, and it directs the flow of data between each part of the design. To make the design simpler, there is one main/top controller which activates other units. The top controller is responsible for configuring the units that are needed in each layer. DMA can move the necessary control data to and from the control unit and can effectively transfer higher levels of data flow control commands while the control unit can manage each resource at a lower level of control.

There is a configuration bus which is responsible for communicating with the main CPU and DL Accelerator. And there are several other control units which are invoked by the configurable bus controller. The control unit has multiple internal counters dividing the data into sections, controlling the flow. Each layer of the neural network algorithm calculation needs its own operations. There are a series of calculations and convolutions done on the data based on the coefficients respective of their widths, heights, and channel numbers, and then there are different kernel numbers, kernel heights, and kernel widths, which further require configuring the resources accordingly. The pooling and activation stages of the data flow also need a control unit to decide. This is done by controlling the signals to stimulate the inputs of different units and latching the outputs.

The dynamic kernel size is utilized to simplify the control unit and decrease the number of various kernels. In this work, the kernel size is 3*3 by default. And, if the kernel size is smaller than the default size, then zero-padding should be added. In other cases, if the kernel size is bigger than the default size, it should be divided into multiple 3*3 matrices.

There are multiple configuration registers inside the control unit which can be configured to control this flow, and as they are being configured through DMA, the control unit communicates with several other smaller control units inside each part of the design, handshaking the data to the next phase and activating them at the right moment. And as this is a time-critical task, there is a dedicated configuration bus for all the control communications.

5.4 Global Buffer

To process multiple neural network layers, there is a need to provide neurons output values from previous layers, therefore the global buffer is allocated to store the input data and weight data. The global DMA transfers the required data from external memory into the global buffer. As this is the most time-consuming process of the accelerator, it reduces the data movement by storing the data inside the buffer. On other hand, as a trade-off Memory on chip consumes huge amount of area and it is costly.

The output of each layer will be the input of the next iteration, and this process is repeated until the last layer, to take advantage of this periodic operation, the result of first computation will be stored in the global buffer and it will be used for the next iteration, it is possible to use the memory page mirroring technique for this multi-step computation. In the final layer the result will be stored in the output buffer. It should be truncated and packed accordingly to be suitable for the next iteration.

The buffers should be designed in the ping pong method to save the time and reduce the latency. The concept is a way to move the data back and forth so that all the resources can use the data without colliding each other's transaction. Using DMA and indicator signals such as data ready, data is transferred without losing any part and the receiving unit will only take the data into use only if the previous unit is done writing to it.

5.5 Aggregation core

The aggregation core consists of an activation and pooling core. Considering that each ML model does something specific, for example, it may be a sequential NN model with five layers that may be a combination of convolution layer, pooling layer, recurrent layer, and so on. We would like to have several to support for different NN models. Therefore, suggested DL accelerator supports linear and nonlinear activation such as Relu, as well as providing support for maximum, minimum, average pooling.

For example, if adapts the design to the MobileNet NN model. The Basic MobileNet model doesn't have the pooling layer, however according to the [62], using the pooling layer in the MobileNet model with a kernel size of 3x3 and stride 2 provides more accuracy and reduces the size of the hidden layer more than common pooling. Furthermore, this feature reduces parameters and the cost of computation. Pooling exists in our proposed architecture because it encapsulates the image to one label which is needed in some applications such as image classification applications.

6. METHODOLOGY

This chapter explains the methods and tools used in this thesis work. The 22 nm ASIC technology is used for implementation. The tools utilized are Visual Studio [63] for writing and editing the code. Modelsim [64] is used for the RTL simulation and finally Cadence [65] tools for ASIC synthesis. Figure 6.1 demonstrates the whole process of doing the work.

6.1 Procedure

This thesis tries to find which algorithm and RTL implementation is better regarding different parameters such as area, power, and performance for the proposed DL accelerator subsystem. Various approaches proposed in other thesis works and literature are compared, to come up with a conclusion on which algorithm is performing better. Different hardware implementation algorithms were implemented and then ran the simulation and synthesis to come up with the conclusion table of the results.

In this thesis work, multipliers and adders are written in SystemVerilog hardware description language. The Visual Studio Code Integrated Development Environment (IDE) with the SystemVerilog extension is very convenient to write and debug the code. After preparing the Hardware Description Language (HDL) known as Register Transfer Level (RTL) code, there is a need to check the functionality of the design. For this reason, the design is simulated by Modelsim, one of the Mentor graphics tools. Various verification methods are used to ensure that the designed hardware is behaviorally correct according to the models which are written in SystemVerilog as well.

The next step after the simulation and verifying the functionality of the design is the actual hardware implementation at a low level and generating the netlist file and back-end tools for the physical implementation. To do the synthesis, a wrapper file is prepared to hold the top-level design and creates two registers in the input and output of the design, and then by running the synthesis flow. Some features of the design are given in the results, which are compared here.

The data is collected from the synthesis tool summary report. Then it is attempted to use different scenarios to collect multiple varying data and then compare them and list

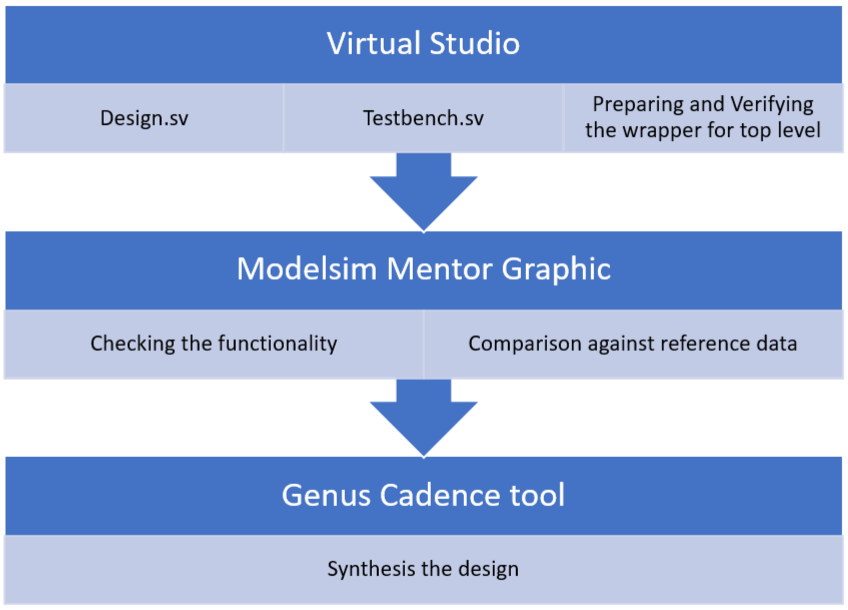

Virtual Studio
Design.sv
Testbench.sv
Preparing and Verifying the wrapper for top level
Modelsim Mentor Graphic
Checking the functionality
Comparison against reference data
Genus Cadence tool
Synthesis the design

Figure 6.1. *Methodology flow*

them in the corresponding table and sort each algorithm based on each feature that has preferred results.

The methodologies used in this thesis follow industry standards and the most elaborate method to verify and choose different architectural implementations in RTL design. By simulating the observed behavior, which is independent of and unaffected by the implementation method and evaluating each design option against it.

6.2 Evaluation

As the field of research is developing and study is carried out related to the ML, DL, NN, and hardware accelerator for specific/general NNs. Therefore, it was important to know how to compare them. These are the metrics for comparing a DLA.

- Operation Per Seconds (OPS) which the unit would be Tera/Giga/Mega operation per second (TOPS, GOPS or MOPs) is the best value for comparing the performance of various DL accelerators. It is possible to include the power into this number and report Operation per second normalized by power consumption (TOPS/W, GOPS/W or MOPs/W). Equation 6.1 demonstrates it.

$$\frac{\text{Operations}}{\text{second}} = \left(\frac{1}{\frac{\text{cycles}}{\text{operation}}} * \frac{\text{cycle}}{\text{second}}\right) * \text{Number of MAC units} * \text{Utilization of MAC units} \tag{6.1}$$

- Throughput demonstrates the computational capability of the hardware. There are effective throughput which obtain in real world conditions, and theoretical throughput that the number of arithmetic units (MAC units) and clock frequency determines it. The achievable throughput relates to the number of utilized MACs. The overall system theoretical throughput:

$$\text{Utilization of MACs} = \frac{\text{Number of active MACs}}{\text{Number of MACs}} * \text{Utilization of active MACs} \quad (6.2)$$

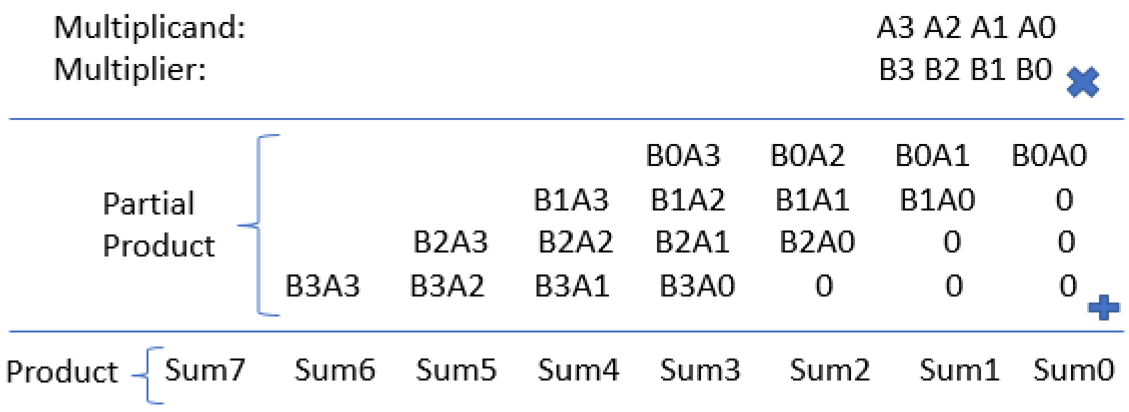
- Latency is the amount of time it takes to return results for given inputs. Low latency is one of the essential criteria for real-time applications such as autonomous navigation.
- Accuracy determines the quality of the results for specific task. The difficulty of the task and dataset affects the accuracy. As an example, performing the classification model on MNIST is less complex than on ImageNet. Object detection is more difficult than classification.
- Energy and power are one of the most critical variables in edge devices. It determines the quality of an edge accelerator.
- Hardware cost such as area of the chip and memory bandwidth
- Flexibility, which refers to the DL accelerator's ability to perform a variety of tasks.
- Scalability refers to the number of different DNN models supported by the DL accelerator.

The accelerator design that can handle multiple NN with the least amount of area and power consumption is preferred. There is a trade-off between throughput, power consumption, precision of weights and activation, and inference accuracy. For example, reducing the power consumption will reduce the inference accuracy as well [24].

7. IMPLEMENTATION OF MULTIPLIERS

There is one multiplier in every MAC unit of the proposed DL architecture. There are four MAC arrays in the proposed DL accelerator. Each MAC array consists of 64 MAC units. Thus, there would be 256 multipliers in total.

Figure 7.1 demonstrates the 4-bit multiplication calculation on pen and paper. The most traditional way of doing multiplication is with addition and shift. The partial products are generated by using AND gate. Then, the result is the addition of the partial products. Thus, it is possible to say that multiplication is repeated addition. The various multiplier algorithms have their unique structure to generate partial products, and they might use special adders to make the computation more optimized.



Multiplicand:					A3	A2	A1	A0
Multiplier:					B3	B2	B1	B0 ✖
Partial Product					B0A3	B0A2	B0A1	B0A0
				B1A3	B1A2	B1A1	B1A0	0
			B2A3	B2A2	B2A1	B2A0	0	0
		B3A3	B3A2	B3A1	B3A0	0	0	0 ➕
Product	Sum7	Sum6	Sum5	Sum4	Sum3	Sum2	Sum1	Sum0

Figure 7.1. *Multiplication of two 4-bit number*

7.1 Adders

The logic circuit design can get simpler by breaking it down into smaller parts, such as half adders and full adders. These kinds of combinational arithmetic circuits are in many architectures. Half adder consists of one AND gate and one XOR gate. Connecting two half adders create the full adder. Figure 7.2 demonstrates the circuit diagram of the half and full adder. For example, a 2-bit Vedic multiplier has two half adders in its architecture. The Wallace multiplier uses several full and half adders.

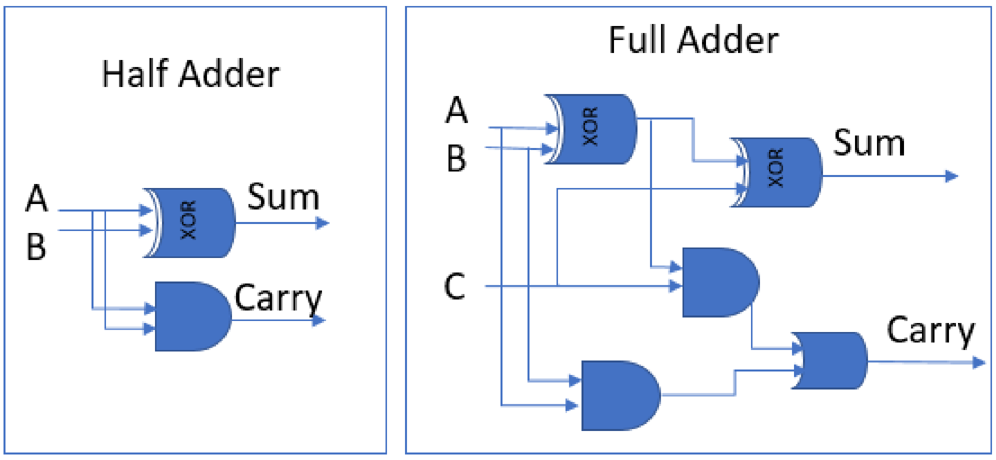


Figure 7.2. *Logic circuit diagram of Half adder and full adder*

Partial products need to be added together in most of the architecture design. Therefore, here is short list and description of different binary adders.

- Ripple Carry Adder (RCA) is built by cascading several full adders. The RCA combines low area usage, high delay time, and higher power consumption.
- Carry Select Adder (CSA) is basically two RCA that are muxed. It does the calculation twice with carry-in zero and one.
- Carry Look Ahead adder (CLA) reduces the propagation delay while it increases the complexity of the HW as bit width increases, but therefore, it is area costly.
- Parallel Prefix Adders (PPA) like Brent-Kung (BK), Kogge-Stone (KS), Han-Charlson (HC) and Lander-Fischer (LF). PPAs are high performance carry tree adders that use the generate and propagate signals. It has main three stages, preprocessing stage, prefix carry stage and post-processing stage. They deliver the best scalability among all adders, but some of them introduce severe routing and fan-out issues.

7.2 Multipliers

The multipliers implemented in this work are listed below along with short description.

- Braun multiplier [66] and Carry Save Array (CSA) Multiplier [67]. These kinds of multipliers are similar to the pen and paper methods. One advantage of them is their regular structure. It is easy to layout. Another advantage is their simplicity of design for a pipelined architecture. In the Braun multiplier first an array of AND gate is used and then an array of half and full adder to produce the final result. while, In CSA multiplier, there is a multiplier adder block array. The AND gate and full adder are coupled to each other and make multiplier adder block.

- Baugh-Wooley multiplier [68], [69] is a technique to use for regular multiplier for two's complement numbers.
- Wallace tree multiplier [70] and Dadda multiplier [71]. These two multipliers are similar, but the difference is in the reduction tree. They both have three stages, generating partial product, row compression and final summation.
- Vedic multiplier [72], [68]. In this multiplier partial products generates vertically and crosswise simultaneously. The final results can produce either with ripple carry adder or parallel prefix adders and maybe combination of different adders.
- Booth multiplier [73] and Modified Booth multiplier [74]. The second one is the improved version of the first. The number of generated partial products is reduced by half in the Modified Booth. Supporting the signed number is the most amazing feature of these multipliers.

The area of most multipliers increases as bit widths increases, except for the booth multiplier, which stays low. Delay and complexity of structure in multipliers vary when bit size grows; for instance, by increasing the bit width of the Array multiplier, the delay increases linearly, and complexity stay low, while the Booth-like multiplier's delay is non-linear, and complexity is medium, moreover, in the Wallace-like multiplier's delay is logarithmic, with high complexity.

All the multiplier algorithms do not support signed multiplication but there are negative weights in most of the ML algorithms and multipliers should be able to calculate the signed multiplication. To support the signed multiplication; the modified two's complement of each input is calculated. Then, based on AND of one and the most significant bit (MSB) of multiplicand and multiplier, which is the sign bit, the multiplexer selects the suitable inputs for the unsigned multiplier, and then after calculating the two's complement of the result. The final product of multiplication is the output of the third multiplexer, which it selects signal is the result of the XOR of the two first multiplexer's select signals.

Figure 7.3 shows how the conversion of the unsigned to the signed multiplier. The idea of making an unsigned multiplier suitable for signed multiplication based on the signed-unsigned multiplier design of [75]. Braun, Wallace, Array and Vedic multiplier are not signed by default, and to calculate the signed number they use structure in the Figure 7.3.

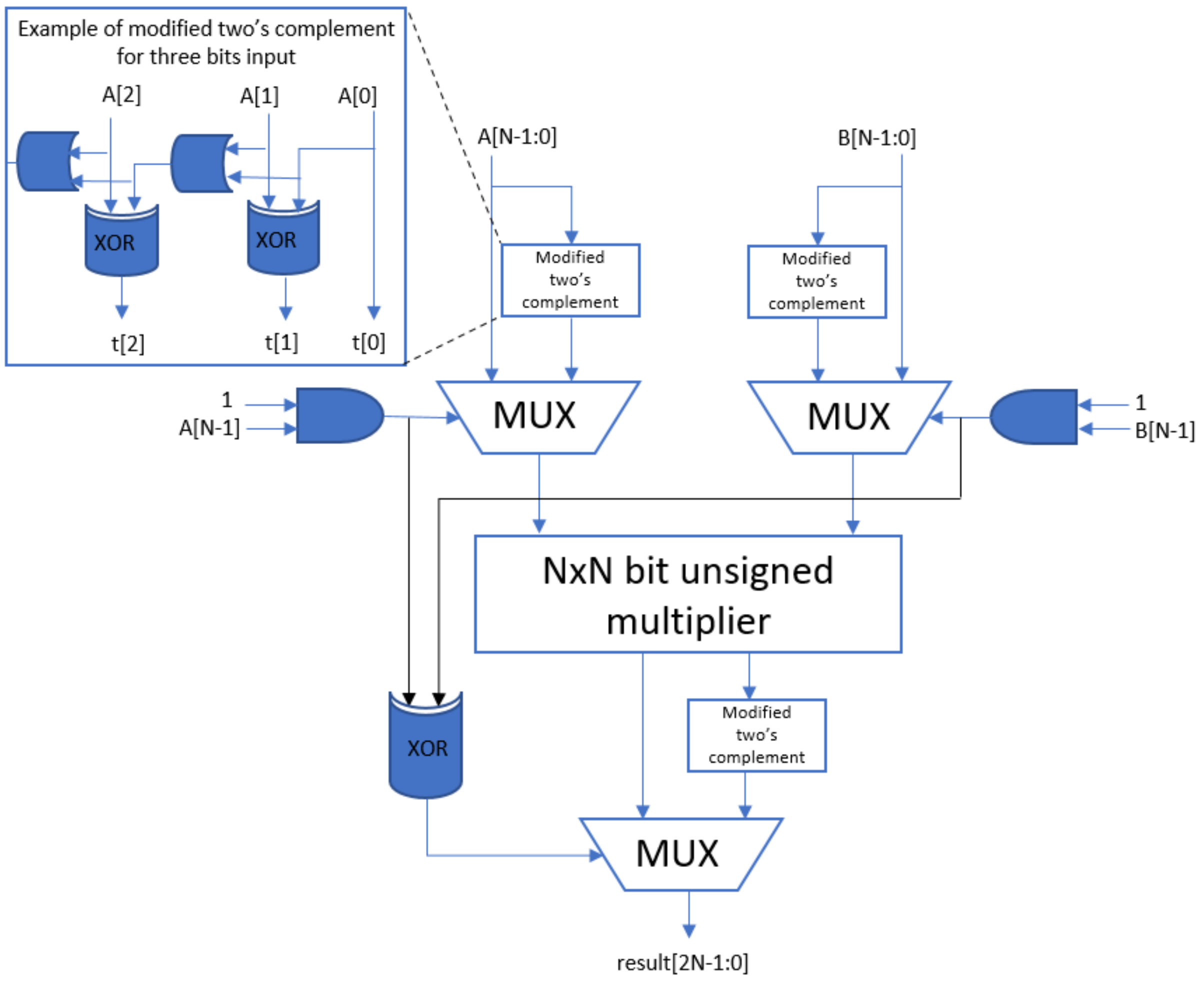

Example of modified two's complement for three bits input
A[2]
A[1]
A[0]
XOR
XOR
t[2]
t[1]
t[0]
A[N-1:0]
B[N-1:0]
Modified two's complement
Modified two's complement
1
A[N-1]
MUX
MUX
1
B[N-1]
NxN bit unsigned multiplier
XOR
Modified two's complement
MUX
result[2N-1:0]

Figure 7.3. *Signed multiplication*

The characteristic of the multiplier is based on adders. For example, there are three adders in an 8-bit Vedic multiplier, and inside each 4-bit multiplier, so there are three 4-bit adders. Figure 7.4 shows the structure of the Vedic multiplier. Thus, different ways of implementing the adder would affect the performance of multipliers. All the adders can be RCA, Parallel prefix adders like Kogge-Stone or Brent-Kung, and eventually, combinations of Brent-Kung, carry save adder, and carry select adder.

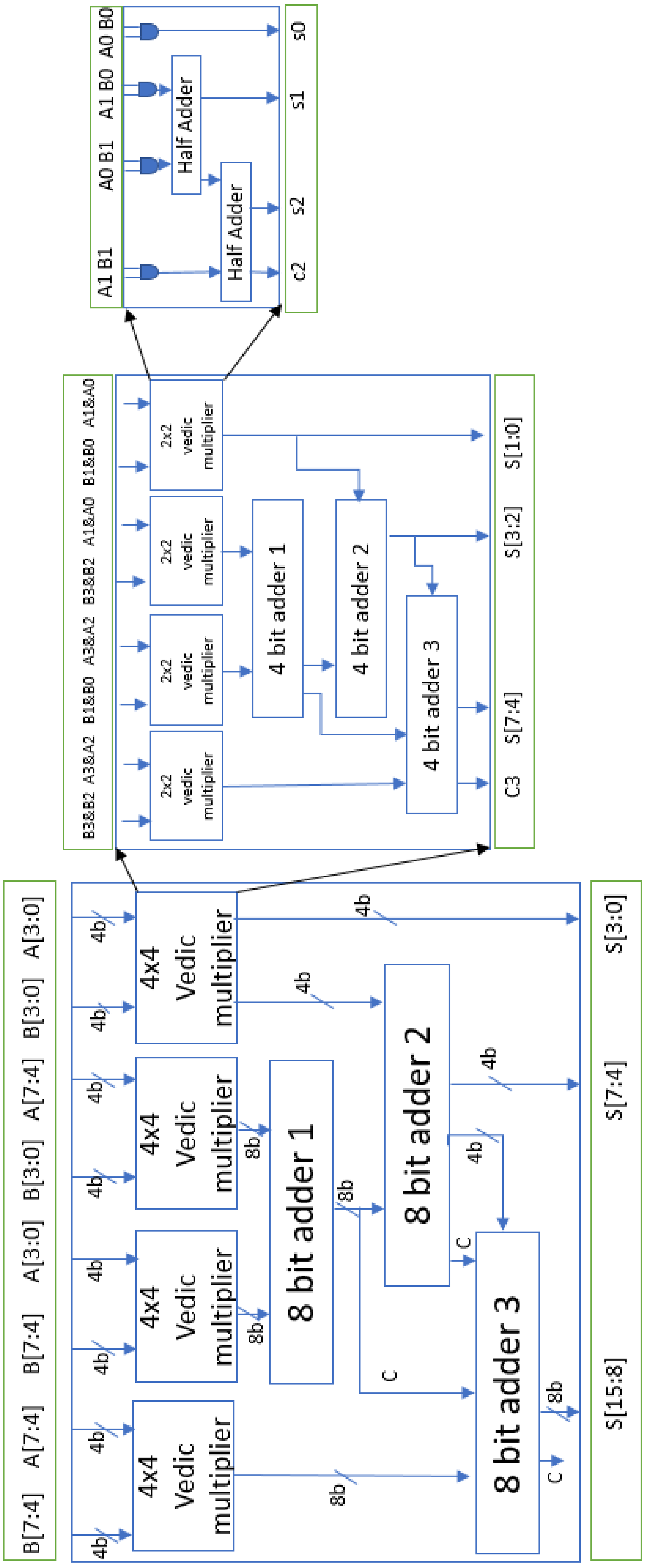


Figure 7.4. *Vedic multiplier*

8. RESULTS AND ANALYSIS

This chapter compares and evaluates every design and implementation based on four criteria: area, power, performance, and average fan-out. Each multiplier and adder are evaluated separately. Since the entire system runs at 1 GHz, the multipliers must at least run at the same speed as the other subsystems.

8.1 Multipliers and Adders

Table 8.1 demonstrates the multipliers and compares them in terms of total area, which includes the cells, physical and net, and total power consumption, which consists of leakage and dynamic power. Target synthesis is 1GHz clock frequency and average fan-out is give separately. A gate output's fan-out is the number of inputs it can support. Fan-out determines how effectively the design is routed, and likelihood of meeting time constraints.

According to Table 8.1, Modified Booth multipliers have the smallest area footprint and lowest power usage. Furthermore, the Baugh-Wooley has the second place in terms of lowest area and power consumption. However, it has the lowest fan-out.

Multiplier	Total Area (um^2)	Total Power (mW)	Latency (ns)	Average Fan-out
Array[1]	1053.996	0.749	1.097	1.8
Modified Booth	531.294	0.476	0.964	1.8
Braun	783.200	0.733	0.990	1.8
Baugh-Wooley	584.909	0.498	0.964	1.4
Vedic-RCA	839.146	0.679	0.984	1.6
Wallace	768.623	0.537	0.585	2.2

Table 8.1. *Area, Power, Latency, and average Fan-out of 8-bit signed multipliers, CLK = 1GHz*

[1] at 960 MHz

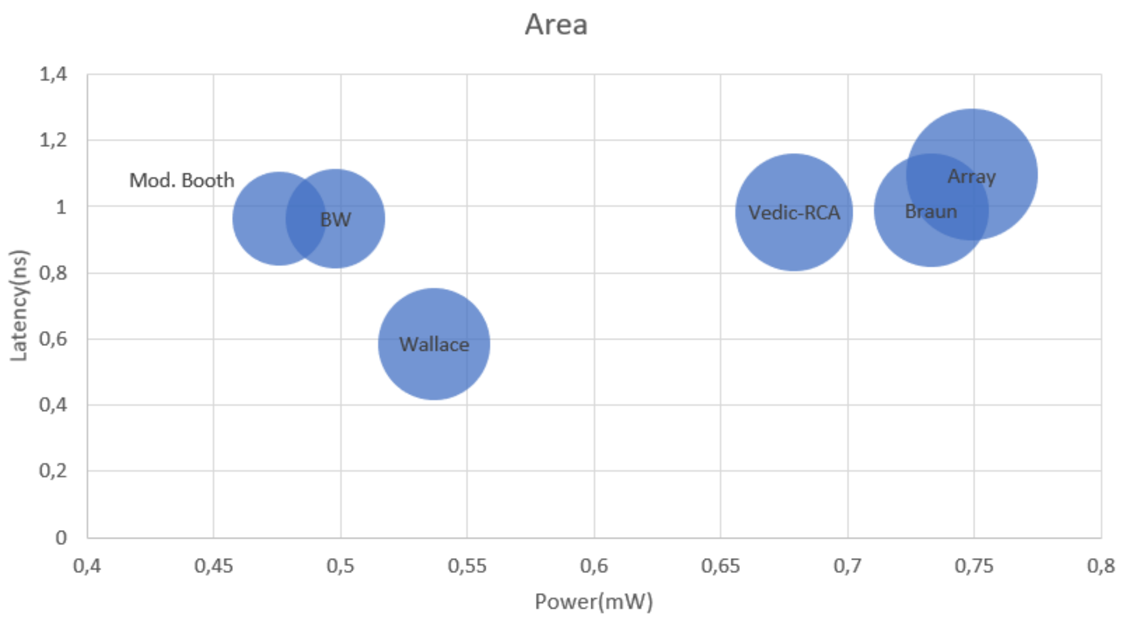


Figure 8.1. *Area, Power, and Latency of Table 8.1 visualized*

Figure 8.1 demonstrates the information in the Table 8.2 as it can be seen here, the size of the bubbles is the area footprint of each multiplier. Array multiplier has the highest area and power consumption, while Modified Booth has the lowest. The Wallace multiplier has the lowest latency, being almost half of the others.

The array multiplier cannot execute at 1 GHz frequency, the Worst Negative Slack (WNS) becomes negative. The results in the table are related to its best performance at a 960 MHz clock frequency. It has the most area and power consumption. As it was expected, the Baugh-Wooley multiplier is an improved version of the array multiplier.

Multiplier	Total Area (um^2)	Total Power (mW)	Latency (ns)	Average Fan-out
Array[1]	58712.847	47.566	1.206	1.7
Modified Booth	41220.807	38.776	0.954	1.8
Braun	69417.560	67.782	0.987	1.8
Baugh-Wooley	42530.191	42.574	0.956	1.5
Vedic-RCA	64506.278	56.268	0.971	1.6
Wallace	47090.312	45.007	0.95	2.2

Table 8.2. *Area, Power, Latency, and average Fan-out of various MAC array, CLK = 1GHz*

[1] at 960 MHz

Table 8.2 illustrates the synthesis result for a MAC array made up of 64 MAC units. Each MAC unit contains one multiplier and one adder. The NoC of the MAC array follows the weight stationary structure. Modified Booth exhibits the best outcomes when compared to other designs. Baugh-Wooley is placed in the second-best performance.

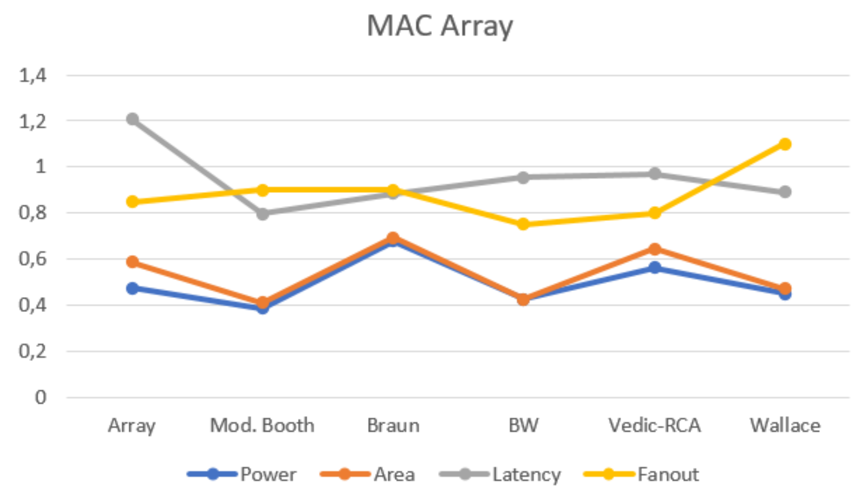


Figure 8.2. *Area, Power, Latency, and Fan-out of Table 8.2 normalized*

Figure 8.2 shows the normalized values of each parameter compared to each other, here the values of area, power, latency, and fan-out are normalized to a value between [0,1] to demonstrate the differences between various multipliers of Table 8.2.

Figure 8.3 compares the performance of the implemented multiplier in terms of CLK frequency. The Modified Booth and Wallace multipliers can work at a 1.5 GHz frequency, which is the same frequency as the entire system and the rest of multipliers at 1 GHz. Modified Booth also has the lowest area and power consumption Table 8.1. Therefore, it is worthy of further development. However, the Baugh-Wooley and Wallace multipliers are not taken into consideration for further development because of their high level of design complexity, especially since their routing systems are too complex to add the flexibility feature to them. Moreover, while the Vedic-RCA multiplier consumes considerable amounts of area and power, and it only works at a 1 GHz frequency, it is modular. Thus, it is feasible to add flexibility to it.

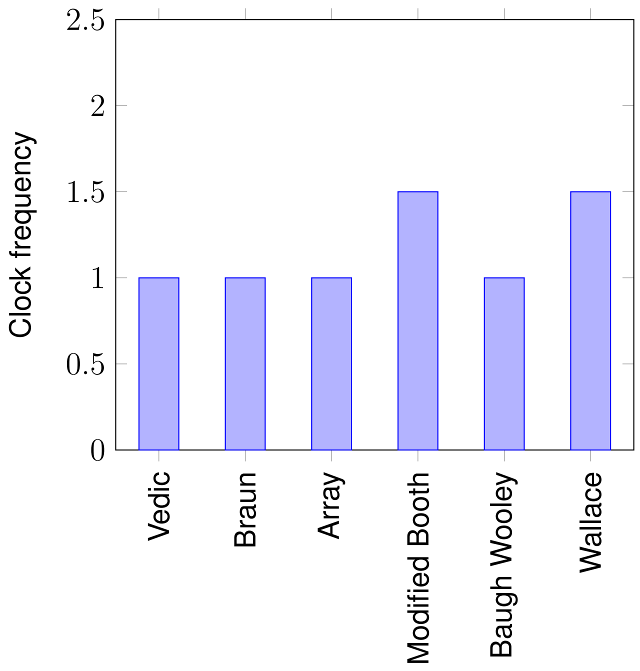


Figure 8.3. *Maximum clock frequency that a multiplier can perform*

Based on the above conclusion, the remainder of the study places a greater emphasis on Modified Booth and Vedic multipliers.

To provide flexibility to the design, it was decided to implement the multipliers with the ability to calculate different bit widths from 2-bit, 4-bit, and 8-bit calculations. However, the input and output bit widths are always 8-bit. However, data in the form of 2-bit and 4-bit can be stored in memory.

Table 8.3 shows two multipliers that support flexible multiplication. It is possible to calculate four multiplications with 2-bit and two multiplications with 4-bit. The structure of the Vedic multiplier is changed to make it flexible as shown in Appendix A.10.

Multiplier	Area (um^2)	Total power (mW)	Latency (ns)	Average Fan-out
Vedic-RCA-Flexible	836.343	0.684	0.961	1.6
Modified-Booth-Flexible	866.461	0.653	0.926	1.9

Table 8.3. *Area, Power, Latency, and average Fan-out bit width multipliers, CLK = 1GHz*

8.2 Exploring different Vedic multiplier implementations

Further optimization was attempted of the Vedic multiplier. According to Figure 7.4, Vedic multiplier has a modular structure. There are several adders inside the Vedic multiplier. Therefore, using a combination of different adders might be useful. This section investigates the performance of the Vedic multiplier using a variety of adders. Table 8.4 presents the area and power of 8-bit adders at 1 GHz clock frequency. The 8-bit CSA has the lowest chip area and power consumption and the highest performance compared to other adders.

Adder	Total area (um^2)	Total power (mW)	Latency (ns)
RCA	136.274	0.144	0.953
BK	129.163	0.136	0.775
PPA	147.963	0.145	0.689
KS	154.033	0.154	0.708
CSA	105.901	0.117	0.363

Table 8.4. *Comparison of different implementation of 8-bit adders, CLK = 1 GHz*

It should be mentioned that, according to [76], when the bit widths increase, passing the carry through different stages will consume more time and delays will become dominant. It should be noted that as bit widths rise, transferring the data through the various steps will take more time, and delays will become more prevalent. Additionally, it distinguishes each distinct adder implementation. When the bit width increases, PPAs perform much better than CLA, RCA, and CSA. However, there is a tiny variation between adders in terms of area efficiency and power consumption for lower bit widths (2-8 bits).

To find out the best optimal design of 8-bit Vedic multiplier, different designs were implemented, investigated, and compared. Tables 8.5 and 8.6 demonstrate the area, and power in 1 GHz clock cycle, respectively. The Vedic multiplier with the Kogge–Stone adder consumes the most area and power, which was expected according to Table 8.4. As the adder alone is area hungry. Similarly, the Vedic multiplier with a combination of BK, CSA and RCA is the most optimal one among the others.

Multiplier	Area (Cell+Physical)	Total area (Cell+Physical+Net)
Vedic-RCA	651.672	839.146
Vedic-BK-CSA-RCA	559.566	729.311
Vedic-KS-RCA	621.936	816.395
Vedic-KS	737.982	971.579
Vedic-BK-CSA	487.872	644.837

Table 8.5. *Area in um^2 consumption of different 8-bit signed Vedic multipliers, CLK = 1GHz*

Multiplier	Total Power (mW)	Latency (ns)
Vedic-RCA	0.679	0.984
Vedic-BK-CSA-RCA	0.594	0.960
Vedic-KS-RCA	0.708	0.961
Vedic-KS	0.734	0.921
Vedic-BK-CSA	0.572	0.956

Table 8.6. *Performance and power in mW consumption of different 8-bit signed Vedic multipliers, CLK = 1GHz*

However, the Vedic multiplier with the combination of Brent-Kung and carry save adder shows a better result than Vedic with ripple carry adder. Yet, the Modified Booth mentioned in 8.1 is better than Vedic in terms of area footprint and power consumption. It is selected as the multiplier implementation for the proposed architecture, and based on our constraints, we sweep through different frequencies of this implementation. Table 8.7 demonstrates the area and power consumption of the signed Modified Booth multiplier at different clock frequencies.

Frequency (GHz)	Total Area (um^2)	Total Power (mW)	Latency (ns)
0.5	479.889	0.215	1.921
1	531.294	0.476	0.964
1.5	590.154	0.795	0.646

Table 8.7. *Area and power of 8-bit signed Modified Booth multipliers at different frequency*

This result is normalized and visualized in Figure 8.4 and, as it can be seen here, by increasing the frequency, total power consumption increases drastically, although the latency reduces, which is expected because of the frequency. Based on this the overall requirement for the system, 1 GHz is the selected frequency.

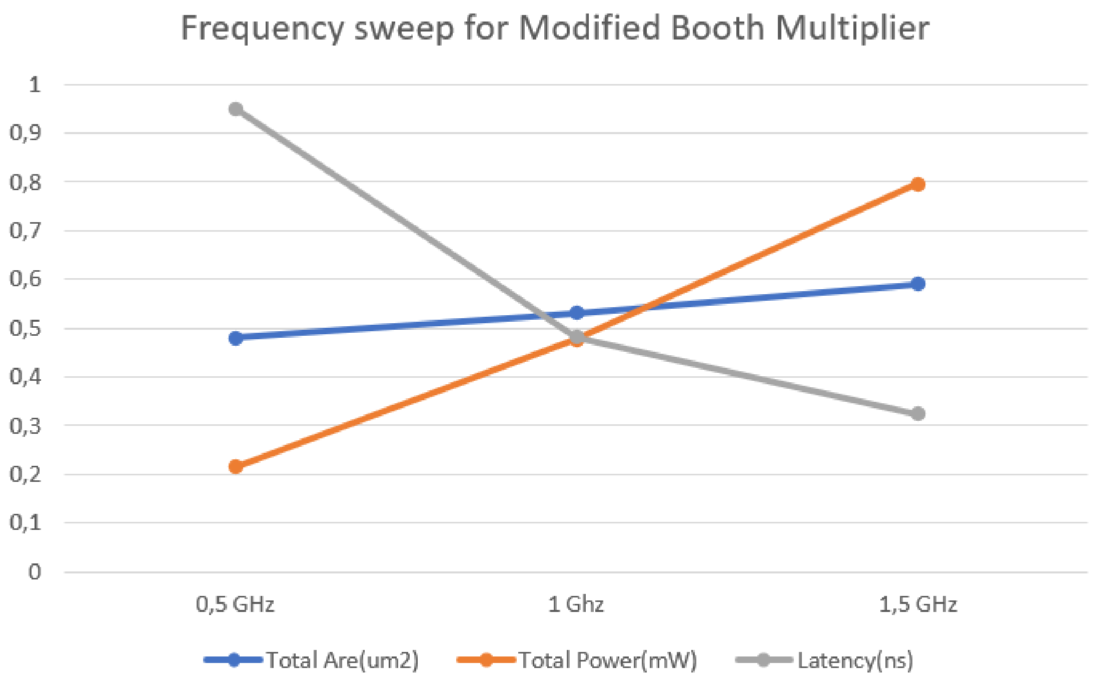


Figure 8.4. *Normalized values of Table 8.7*

8.3 Performance analysis of proposed deep learning accelerator

Performance should be pointed out, as it helps to compare various designs. However, it should be noted that deep learning accelerators are evaluated based on a variety of factors, as detailed in the section 6.2. Here the performance of the proposed architecture is evaluated. The performance of the multipliers is available, but the other IP's performances are estimated. The reason for evaluating the performance is to find the bottlenecks in the proposed DL architecture. The aim is to improve it in the future works.

A NN model contains several layers, and each layer's neurons are repeated several times until features are extracted. The considered NN model is MobileNet, as described by python code in the Listing 8.1. The model is trained on the ImageNet dataset. Figure 8.5 shows the basic structure of the NN model. The calculation is only for the first layer, which is convolution and is shown with the orange circle.

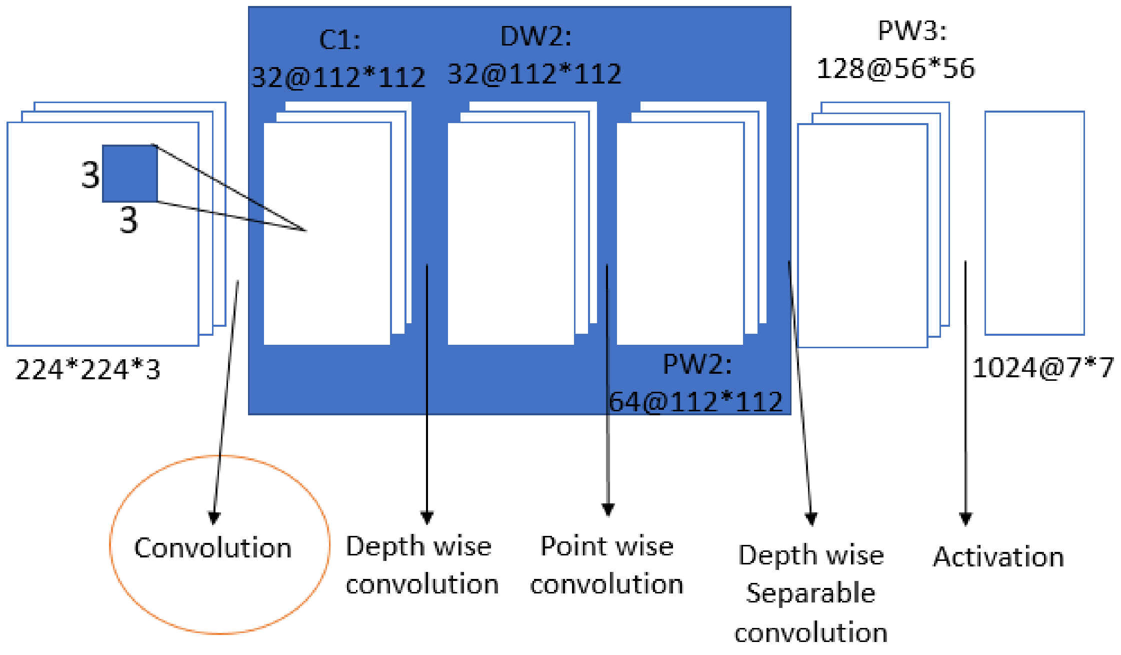


Figure 8.5. *Basic illustration of simple MobileNet layer*

Listing 8.1. *Python code of a simple MobileNet model*

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing import image
import matplotlib.pyplot as plt
from tensorflow.keras.applications import imagenet_utils
from IPython.display import Image

# importing image
filename = '/content/test.jpg'

Image(filename,width=224,height=224)
img = image.load_img(filename,target_size=(224,224))

#initializing the model to predict the image details using predefined models.
tf.keras.applications.mobilenet.MobileNet(
    input_shape=None,
    alpha=1.0,
    depth_multiplier=1,
    dropout=0.001,
    include_top=True,
    weights='imagenet',
    input_tensor=None,
    pooling=None,
    classes=1000,
    classifier_activation='relu',
    **kwargs)

resizedimg = image.img_to_array(img)
finalimg = np.expand_dims(resizedimg,axis=0)
finalimg = tf.keras.applications.mobilenet.preprocess_input(finalimg)
finalimg.shape
predictions = model.predict(finalimg)

results = imagenet_utils.decode_predictions(predictions)
print(results)
```

The model is mapped to the proposed DLA on pen and paper. For this model, DMA, top control unit, CNN core, and activation are enabled, and data goes through these IPs shown in Figure 8.6 with stars. There are two types of data: control data (green arrows), and computation data (orange arrows). Control data is transferred to the top controller. Then it activates different IPs based on the given values.

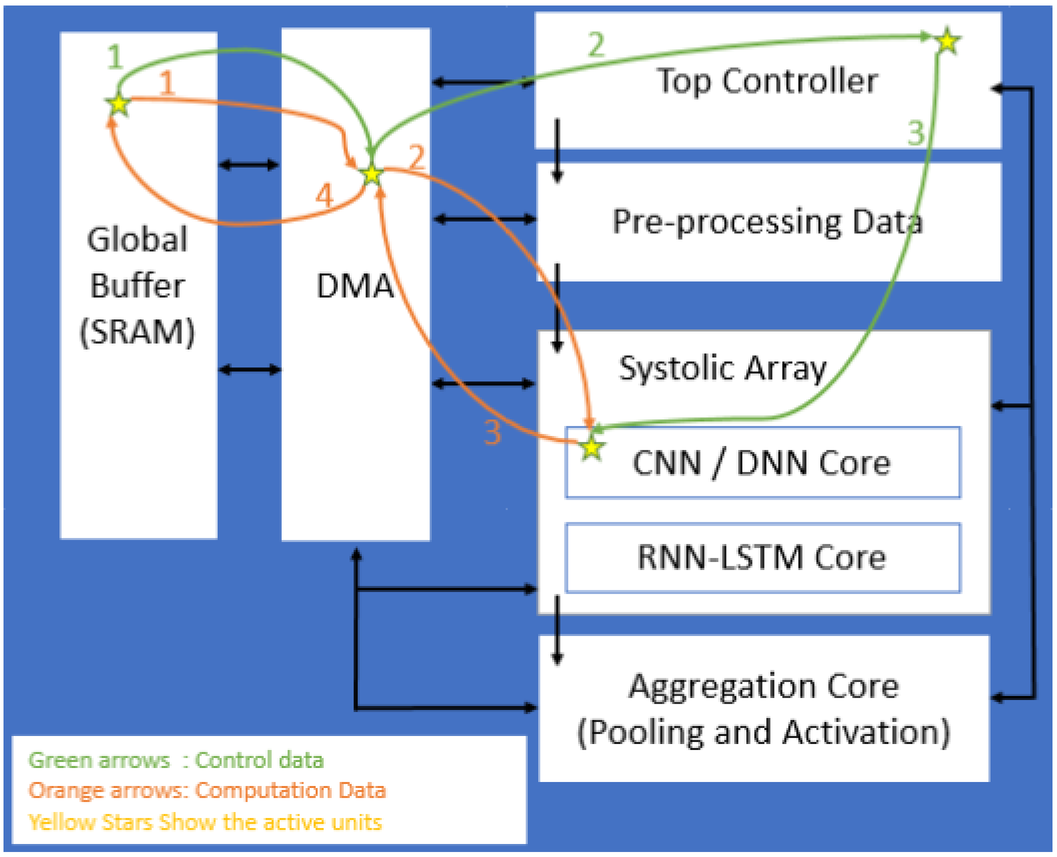

Global Buffer (SRAM)
DMA
Top Controller
Pre-processing Data
Systolic Array
CNN / DNN Core
RNN-LSTM Core
Aggregation Core (Pooling and Activation)
1
1
2
2
3
3
4
Green arrows : Control data
Orange arrows: Computation Data
Yellow Stars Show the active units

Figure 8.6. *Data movements between IPs for MobileNet NN model*

Based on the assumption that the input image size is 224*224*3. If we assume the weight dimension, in other words, the kernel size is 3*3, then there will be 150528 multiplications needed for the first layer of convolution. In each MAC array, there are 64 multipliers, and there are 4 MAC arrays in the proposed architecture, which makes the number of multipliers 256 in total. Then, for a specified layer, the systolic array should be executed 588 times (150528 multiplications divided by 256 multipliers = 588). This step is shown by the green square in the Figure 8.7, each MAC operation would take 3 clk cycles (1 cycle multiply + 1 cycle addition + 1 cycle writing the data) which means 1764 cycles in total.

Then, from a hardware point of view, we set the clock speed for the DL accelerator at 1 GHz. The signed Modified Booth multiplier takes 0.964 ns to execute. However, as the multipliers are working in parallel, and since the system clock is fixed to 1GHz, the overall time used for calculating the convolution is 588* 3 * 1ns = 1764 ns for C1. look at the Figure 8.5 for better understanding.

A multi-cycle control unit has four operations that will be completed in multiple stages. Considering the timing of the DMA, which is responsible for distributing data in the systolic array, it has a minimum of four steps: fetch instruction, decode instruction, fetch operand, and execute the instruction. Besides these, it must store the results and process the interrupts as well.

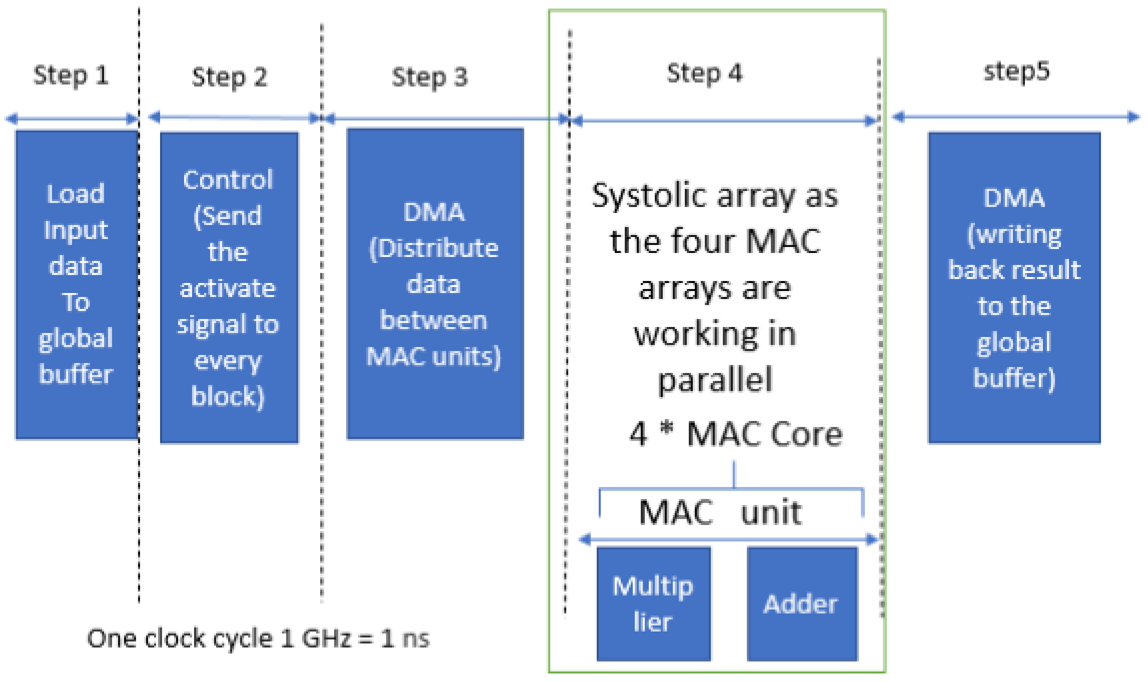

Step 1
Step 2
Step 3
Step 4
step5
Load Input data To global buffer
Control (Send the activate signal to every block)
DMA (Distribute data between MAC units)
Systolic array as the four MAC arrays are working in parallel
4 * MAC Core
MAC unit
Multip lier
Adder
DMA (writing back result to the global buffer)
One clock cycle 1 GHz = 1 ns

Figure 8.7. *The latency to execute one layer*

Considering using the AXI protocol for all transactions with a data width of 32-bits, every single transaction takes 6 clock cycles according to [77] and can transfer 4 bytes of data. Then, the burst mode is used when transferring the main data, which would take 6 clock cycles + (X amount of 32-bit data). For the 224*224*3, each pixel is one byte or 8-bits, and it means there are 150528 bytes to transfer. This fits into the 256 KB SRAM for one layer. This AXI transaction would require 150528 / 4 = 37632 and 37632 + 6 = 37638 clock cycles to move all the data for transactions number 1 and 2 in Figure 8.6. Then, taking the read delay of SRAM into account, which is 2 clock cycles per read operation, this would sum up to 75264 clock cycles read delay + 6 + 37632 = 112902 clock cycles just for the read from the internal buffer. For transactions numbers 3 and 4, the result after the C1 phase is 32*112*112. Transferring this data back to the buffer would consume 32*112*112 = 401408/4 = 100352 and 100352 + 6 = 100358 clock cycles. Each control data transaction takes 6 clock cycles with 3 phases of control data. For this layer, each control data would take 8 times the 32-bit transfer. This would sum up to 8*6 = 48 clock cycles for control data.

DMA distributes data to the systolic array and writes data back to the memory. This happens in parallel to MAC cores starting their process. The result (32*112*112 or 401408 bytes) of the first layer needs to be written back to the buffer. Although the global SRAM buffer is only 256 KB. This means some part of the data needs to be sent back to the main memory or the DRAM, as only half of those 401408 bytes would be left out. The calculation is based on the AXI transaction from the systolic array to the buffer via DMA. There are 401408 bytes / 4 = 100352, and the write operation to SRAM has a 6 clock cycle delay per operation. Thus, 6*401408 + 100352 + 6 = 2508806. Assuming that read and write operations to the DRAM have an 80 clock cycle delay for each transaction for our remaining 200704 bytes, this would mean 200704*80 = 16056320 clock cycles delay, and again, transferring this data with the AXI interface would take 200704/4 = 50176, 16056320 + 50176 = 16106496 clock cycles.

Finally, as we consider only this layer of MobileNet execution, it takes 112902 (clock cycles read operation) + 1764 (clock cycles MAC operations) + 2508806 (cycles write operation) + 16106496 (all the DRAM transactions) + 48 (control data) = 18730016 clock cycles, this means 18730016 ns at the 1 GHz clock.

It is noteworthy to mention OPS/W for the first layer of MobileNet in the proposed DL accelerator. There are 256 MACs in the proposed DLA. By considering the full utilization, according to equation. 6.1, we now know that one operation for one layer takes 18730016 ns which mean each cycle we complete 1 / 18730016 = 5.339023e-8 operations, the OPS would be calculated as such:

$$\frac{\text{Operations}}{\text{second}} = \left(\frac{1}{\frac{\text{cycles}}{\text{operation}}} * \frac{\text{cycle}}{\text{second}}\right) * \text{Number of MAC units} * \text{Utilization of MAC units} \tag{8.1}$$

$=$ 5.339023e-8 * 256 * 100 $=$ 0.001366 TOPS = 1.366 GOPS = 1366 MOPS

Hence, the power consumption of all IPs are not available, only GOPS/W for the systolic array is reported. The power consumption of one MAC array is 38.776 mW and there are four of them, so in total, the power consumption of the systolic array is 0.155W (38.776*4 =155,104 mW). Therefore, the GOPS/W of the systolic array for calculating the assumed layer of NN would be 1.366 GOPS/W. This calculation is only for the systolic array and not the entire DL accelerator subsystem, although most of the reported TOPS/W in other articles are based on the entire NN model calculation and entire chip power consumption, therefore they are not one-to-one comparable. Moreover, some calculation and assumptions are made which are not cycle accurate.

8.4 Comparison of this work with related works

A few related papers have reported metrics of layer-by-layer computation of one NN model on their accelerator. For example, Eyeriss [35] reports the metrics of their DL accelerator based on the AlexNet model. According to the author, for the first layer of AlexNet, Eyeriss MOPS would be 421.6 at 200 MHz. This work achieves 1366 MOPS at 1GHz for the first layer of MobileNet. Another example would be CENNA which reports based on VGG16 model. In the future work, the ALexNet and VGG16 model is going to be mapped to the suggested architecture to make accurate comparisons.

The focus of thesis was mostly on implementing the systolic array, and how to find a suitable multiplier to save area and power. Bert-Marian [33] and CENNA[28] are similar works. Table 8.8 demonstrates some of their features. According to table 3.2 shows the best results compare to other related works. Comparing this work with them predicts that the final result would be suitable.

Platform	This work[1]	Bert-Marian [33]	CENNA [28]	Eyeriss[35]
Technology (nm)	22	40	65	65
Area (mm^2)	0.256[2]	2.4	1.38	12.25
Power (mW)	101,25[3]	232[4]	47.344	332
Number of MACs	256	256	56 Mul/160 Add	168
Precision (bit)	2/4/8	4/8	16	16
Frequency (MHz)	500	12-204	500	200
Memory (SRAM, KB)	256	144	64	108
Performance (GOPS)	1.366[6]	102[5]	86 [5]	0.4216[6]

Table 8.8. *The comparison of proposed DL Accelerator with other*

[1]MAC unit with Flexible Modified Booth multiplier synthesised at 500 MHz
[2]Area of systolic array without considering the area consumption of other IPs
[3]Only power consumption of systolic array without considering the power consumption of other IPs
[4]MAC array + (MEM + Control = 42mW) = 274mW
[5]Peak Performance of entire DL accelerator
[6]Peak Performance for one layer of ML model

9. CONCLUSIONS

This thesis consists of two main parts. Firstly, various multipliers are implemented and evaluated, which are the most critical units in the deep learning subsystem. To determine the most efficient ones regarding power, area, and latency. As a result, a flexible multi-precision (2,4,8 bit) signed Modified Booth multiplier is selected. It can perform fast multiplication with the lowest chip area and power consumption compared to the others.

Secondly, it proposes a flexible deep learning accelerator that utilizes the selected multiplier and supports CNN. The proposed architecture is scalable enough to support different bit widths (2, 4, and 8 bits). Furthermore, the focus was to reduce the size of the arithmetic unit in the DL subsystem based on the first part of the thesis and evaluate its effect on the proposed DL accelerator. The systolic array contains 256 multipliers, and the area and power are reduced by using the Modified Booth multiplier.

Moreover, since the designed DL accelerator is part of a bigger system, it is important to consider the communication between different subsystems. If the multipliers have lower latency, they need to stay idle until the next data is ready. Therefore, it is pointless for the intended multiplier to be faster than the entire design's critical path.

The suggested deep learning accelerator has two key characteristics. In addition to using less area, more energy-efficient systolic array, it maintains data transfer efficiency. This proposed DL accelerator is designed based on the Ballast chip architecture. It inherited the same input and output interfaces. One of the future tasks is to replace the existing Ballast Deep Learning accelerator by the proposed one and evaluate its performance in a large chip implementation. On the reported area footprint and power usage, it seems to be clear improvement. Furthermore, we will evaluate more CNNs and other NN models to figure out how they can be optimally mapped to the proposed DL accelerator.

REFERENCES

[1] Rautakoura, A., Hamalainen, T., Kulmala, A., Lehtinen, T., Duman, M. and Ibrahim, M. Ballast: Implementation of a Large MP-SoC on 22nm ASIC Technology. *Proceedings of the IEEE* (2022).

[2] Muhammad, I. and Yan, Z. SUPERVISED MACHINE LEARNING APPROACHES: A SURVEY. *ICTACT Journal on Soft Computing* 5.3 (2015).

[3] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[4] Krizhevsky, A., Sutskever, I. and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM* 60.6 (2017), pp. 84–90.

[5] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[6] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[7] He, K., Zhang, X., Ren, S. and Sun, J. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[8] Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. *European conference on computer vision*. Springer. 2014, pp. 818–833.

[9] Sze, V., Chen, Y.-H., Yang, T.-J. and Emer, J. S. *Efficient Processing of Deep Neural Networks*. Morgan Claypool publishers, 2020, pp. 1–341. ISBN: 9781681738314.

[10] Krizhevsky, A., Sutskever, I. and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[11] Iandola, F. N., Moskewicz, M. W., Ashraf, K. and Keutzer, K. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2592–2600.

[12] Xie, S., Girshick, R., Dollár, P., Tu, Z. and He, K. Aggregated residual transformations for deep neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500.

[13] Chollet, F. Xception: Deep learning with depthwise separable convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[14] Howard, A., Pang, R., Adam, H., Le, Q., Sandler, M., Chen, B., Wang, W., Chen, L.-C., Tan, M., Chu, G. et al. Searching for MobileNetV3. 1314–1324. *DOI: https://doi. org/10.1109/ICCV* (2019).

[15] Singh, S. and Krishnan, S. Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 11237–11246.

[16] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[17] *Deep Learning Training vs. Inference: What's the Difference? (xilinx)*. https://www.xilinx.com/applications/ai-inference/difference-between-deep-learning-training-and-inference.html. Accessed: 2022-10-07.

[18] Wikipedia contributors. *TensorFlow — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=TensorFlow&oldid=1096706403. [Online; accessed 20-July-2022]. 2022.

[19] Wikipedia contributors. *PyTorch — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=PyTorch&oldid=1098134239. [Online; accessed 20-July-2022]. 2022.

[20] Wikipedia contributors. *Keras — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Keras&oldid=1090965329. [Online; accessed 20-July-2022]. 2022.

[21] Wikipedia contributors. *Caffe (software) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Caffe_(software)&oldid=1093626944. [Online; accessed 20-July-2022]. 2022.

[22] Valavi, H., Ramadge, P. J., Nestler, E. and Verma, N. A 64-Tile 2.4-Mb In-Memory-Computing CNN Accelerator Employing Charge-Domain Compute. *IEEE Journal of Solid-State Circuits* 54.6 (2019), pp. 1789–1799. DOI: 10.1109/JSSC.2019.2899730.

[23] Nurvitadhi, E., Sim, J., Sheffield, D., Mishra, A., Krishnan, S. and Marr, D. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577314.

[24] *Advances in Computers. Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*. Zoe Kruze, 2021, pp. 1–402. ISBN: 978-0-12-823123-4.

[25] A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development.

2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.

[26] Chen, Y.-H., Yang, T.-J., Emer, J. and Sze, V. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[27] Chen, Y.-H., Krishna, T., Emer, J. S. and Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[28] Park, S.-S. and Chung, K.-S. CENNA: Cost-Effective Neural Network Accelerator. *Electronics* 9.1 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9010134. URL: https://www.mdpi.com/2079-9292/9/1/134.

[29] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A. and Dally, W. J. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 243–254. DOI: 10.1109/ISCA.2016.30.

[30] Chang, K.-W. and Chang, T.-S. VWA: Hardware Efficient Vectorwise Accelerator for Convolutional Neural Network. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.1 (2020), pp. 145–154. DOI: 10.1109/TCSI.2019.2942529.

[31] Luo, T., Liu, S., Li, L., Wang, Y., Zhang, S., Chen, T., Xu, Z., Temam, O. and Chen, Y. DaDianNao: A neural network supercomputer. *IEEE Transactions on Computers* 66.1 (2016), pp. 73–88.

[32] Zhou, S., Guo, Q., Du, Z., Liu, D., Chen, T., Li, L., Liu, S., Zhou, J., Temam, O., Feng, X., Zhou, X. and Chen, Y. ParaML: A Polyvalent Multicore Accelerator for Machine Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.9 (2020), pp. 1764–1777. DOI: 10.1109/TCAD.2019.2927523.

[33] Moons, B. and Verhelst, M. A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. 2016, pp. 1–2. DOI: 10.1109/VLSIC.2016.7573525.

[34] *NVIDIA Deep Learning Accelerator (NVDLA)*. http://nvdla.org/. Accessed: 2022-03-29.

[35] Chen, Y.-H., Krishna, T., Emer, J. S. and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138.

[36] Aimar, A., Mostafa, H., Calabrese, E., Rios-Navarro, A., Tapiador-Morales, R., Lungu, I.-A., Milde, M. B., Corradi, F., Linares-Barranco, A., Liu, S.-C. et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE transactions on neural networks and learning systems* 30.3 (2018), pp. 644–656.

[37] Liu, B., Zou, D., Feng, L., Feng, S., Fu, P. and Li, J. An fpga-based cnn accelerator integrating depthwise separable convolution. *Electronics* 8.3 (2019), p. 281.

[38] Shin, D. and Yoo, H.-J. The heterogeneous deep neural network processor with a non-von Neumann architecture. *Proceedings of the IEEE* 108.8 (2019), pp. 1245–1260.

[39] Shin, D., Lee, J., Lee, J. and Yoo, H.-J. 14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2017, pp. 240–241.

[40] Lee, J., Kim, C., Kang, S., Shin, D., Kim, S. and Yoo, H.-J. UNPU: A 50.6 TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2018, pp. 218–220.

[41] Yin, S., Ouyang, P., Tang, S., Tu, F., Li, X., Liu, L. and Wei, S. A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications. *2017 Symposium on VLSI Circuits*. IEEE. 2017, pp. C26–C27.

[42] Zeng, S., Guo, K., Fang, S., Kang, J., Xie, D., Shan, Y., Wang, Y. and Yang, H. An efficient reconfigurable framework for general purpose cnn-rnn models on fpgas. *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*. IEEE. 2018, pp. 1–5.

[43] Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F. K. and Benini, L. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: `10.1109/TVLSI.2017.2654506`.

[44] Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-first AAAI conference on artificial intelligence*. 2017.

[45] Camus, V., Mei, L., Enz, C. and Verhelst, M. Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.4 (2019), pp. 697–711. DOI: `10.1109/JETCAS.2019.2950386`.

[46] Shin, D., Lee, J., Lee, J. and Yoo, H.-J. 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 240–241. DOI: `10.1109/ISSCC.2017.7870350`.

[47] Chen, Y.-H., Emer, J. and Sze, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 367–379. DOI: `10.1109/ISCA.2016.40`.

[48] Zhou, G., Zhou, J. and Lin, H. Research on NVIDIA Deep Learning Accelerator. *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 2018, pp. 192–195. DOI: `10.1109/ICASID.2018.8693202`.

[49] Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P. luc, Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E. and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ISCA*. ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. URL: `http://dblp.uni-trier.de/db/conf/isca/isca2017.html#JouppiYPPABBBBB17`.

[50] Cavigelli, L. and Benini, L. Origami: A 803-GOp/s/W convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* 27.11 (2016), pp. 2461–2475.

[51] Bernardo, P. P., Gerum, C., Frischknecht, A., Lübeck, K. and Bringmann, O. UltraTrail: A Configurable Ultralow-Power TC-ResNet AI Accelerator for Efficient Keyword Spotting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4240–4251. DOI: `10.1109/TCAD.2020.3012320`.

[52] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y. and Temam, O. ShiDianNao: Shifting vision processing closer to the sensor. *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 92–104. DOI: `10.1145/2749469.2750389`.

[53] Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W. and Dally, W. J. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news* 45.2 (2017), pp. 27–40.

[54] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y. and Temam, O. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. Ed. by R. Balasubramonian, A. Davis and S. V. Adve. ACM, 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: `10.1145/2541940.2541967`. URL: `http://doi.acm.org/10.1145/2541940.2541967`.

[55] Williams, S., Waterman, A. and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 52.4 (2009), pp. 65–76.

[56] Sze, V., Chen, Y.-H., Yang, T.-J. and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.

[57] Mei, L., Dandekar, M., Rodopoulos, D., Constantin, J., Debacker, P., Lauwereins, R. and Verhelst, M. Sub-Word Parallel Precision-Scalable MAC Engines for Efficient Embedded DNN Inference. *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2019, pp. 6–10. DOI: 10.1109/AICAS.2019.8771481.

[58] Seznec, M., Gac, N., Orieux, F. and Sashala Naik, A. Computing large 2D convolutions on GPU efficiently with the im2tensor algorithm. *Journal of Real-Time Image Processing* (2022), pp. 1–13.

[59] Andersson, F. and Beylkin, G. The fast Gauss transform with complex parameters. *Journal of Computational Physics* 203.1 (2005), pp. 274–286. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2004.07.020. URL: https://www.sciencedirect.com/science/article/pii/S0021999104003274.

[60] Motey, Y. M. and Panse, T. G. Hardware implementation of truncated multiplier based on multiplexer using FPGA. *2013 International Conference on Communication and Signal Processing*. 2013, pp. 401–404. DOI: 10.1109/iccsp.2013.6577083.

[61] Pietras, M. Hardware conversion of neural networks simulation models for neural processing accelerator implemented as FPGA-based SoC. *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–4.

[62] Chen, H.-Y. and Su, C.-Y. An Enhanced Hybrid MobileNet. *2018 9th International Conference on Awareness Science and Technology (iCAST)*. 2018, pp. 308–312. DOI: 10.1109/ICAwST.2018.8517177.

[63] *Visual Studio: IDE and Code Editor for Software Developers and Teams*$_2022$. Oct. 2022. URL: https://visualstudio.microsoft.com/.

[64] *ModelSim HDL simulator*. URL: https://eda.sw.siemens.com/en-US/ic/modelsim/.

[65] *Genus Synthesis Solution | Cadence*. URL: https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[66] Rongali, V. K. and Srinivas, B. Design of area efficient high speed parallel multiplier using low power technique on 0.18 um technology. *Int J Advan Res Comput Eng Technol (IJARCET)* 2 (2013).

[67] Habibi, A. and Wintz, P. Fast Multipliers. *IEEE Transactions on Computers* C-19.2 (1970), pp. 153–157. DOI: 10.1109/T-C.1970.222881.

[68] Poornima, M., Patil, S. K., Shivukumar, S. K. and Sanjay, H. Implementation of multiplier using Vedic algorithm. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 2.6 (2013), pp. 219–223.

[69] Sjalander, M. and Larsson-Edefors, P. High-speed and low-power multipliers using the Baugh-Wooley algorithm and HPM reduction tree. *2008 15th IEEE International Conference on Electronics, Circuits and Systems*. 2008, pp. 33–36. DOI: 10.1109/ICECS.2008.4674784.

[70] Sankar, D. R. and Ali, S. A. Design of Wallace tree multiplier by Sklansky adder. *International Journal of Engineering Research and Applications (IJERA)* 3.1 (2013), pp. 1036–1040.

[71] Manu, V., Vijaya Prakash, A. M. and Chandra, M. U. Design and Implementation of Sixteen-bit Low Power and Area Efficient Dadda Multiplier. *2019 4th International Conference on Recent Trends on Electronics, Information, Communication Technology (RTEICT)*. 2019, pp. 631–636. DOI: 10.1109/RTEICT46194.2019.9016834.

[72] Rao, K. D., Gangadhar, C. and Korrai, P. K. FPGA implementation of complex multiplier using minimumd delay Vedic real multiplier architecture. *2016 IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON)*. 2016, pp. 580–584. DOI: 10.1109/UPCON.2016.7894719.

[73] Kalaiyarasi, D. and Saraswathi, M. Design of an Efficient High Speed Radix-4 Booth Multiplier for both Signed and Unsigned Numbers. *2018 Fourth International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*. 2018, pp. 1–6. DOI: 10.1109/AEEICB.2018.8480959.

[74] Kim, S. and Cho, K. Design of high-speed modified booth multipliers operating at GHz ranges. *World academy of science, Engineering and Technology* 61 (2010), pp. 1–4.

[75] Ganjikunta, G. K., Khan, S. I. and Basha, M. M. A High-Performance Signed-Unsigned Multiplier Using Vedic Mathematics. *Journal of Low Power Electronics* 15.3 (2019), pp. 302–308.

[76] Neeraja, B. and Goud, R. S. P. Design of an Area Efficient Braun Multiplier using High Speed Parallel Prefix Adder in Cadence. *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. 2019, pp. 1–5. DOI: 10.1109/ICECCT.2019.8869307.

[77] AMBA, A. AMBA 4 Advanced eXtensible Interface (AXI4) Protocol Specification. *ARM IHI D* 22 (2011), p. 147.

APPENDIX A: APPENDIX

A.1 SystemVerilog Codes

A.1.1 Modified Booth

Listing A.1. *Modified Booth multiplication*

```
/*
* Name: modified_booth.sv
* Date: 16_05_2022
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description :
* Algorithm: (for unsigned numbers)
* 1) Pad the LSB with one zero.
* 2) Pad the MSB with 2 zeros if n is even and 1 zero if n is odd.
* 3) Divide the multiplier into overlapping groups of 3-bits.
* 4) Determine partial product scale factor from modified
booth 2 encoding table.
* 5) Compute the Multiplicand Multiples
* 6) Sum Partial Products

* Algorithm Extension: (for signed multiplier)
* 1) Pad the LSB with one zero.
* 2) If n is even dont pad the MSB ( n/2 PPs ) and
* if n is odd sign extend the MSB by 1 bit ( n+1/2 PPs ).
* 3) Divide the multiplier into overlapping groups of 3-bits.
* 4) Determine partial product factor from table. (booth_recording.v)
* 5) Compute the Multiplicand Multiples
* 6) Sum Partial Products

*/
module modified_booth #(
        parameter int BIT_WIDTH = 8 )
    (   input logic signed  [BIT_WIDTH-1:0] Data_in_A,Data_in_B,
        output logic signed [(2*BIT_WIDTH)-1:0] Data_out_sum
    );
```

```
logic [BIT_WIDTH+1:0] paded_Data_in_B;
logic [2:0] D_B_three_bit;
logic [BIT_WIDTH-1:0][1:0] op;
logic sign_bit;


logic       [10 :1] ps0, ps1, ps2, ps3;
logic       [8  :1]  s1, s2, s3;
logic       [8  :1]  c1, c2, c3;
logic [(2*BIT_WIDTH)-1:0] A,B;
logic cout; // from adder

booth_encoder       inst1   (1'b0, Data_in_B[0],
Data_in_B[1], xP10, xM10, xP20, xM20);

booth_decoder       inst2   (xP10, xM10, xP20,
xM20, Data_in_A[7], Data_in_A[7], temp1);
booth_decoder       inst3   (xP10, xM10, xP20,
xM20, Data_in_A[7], Data_in_A[6], ps0[9]);
booth_decoder       inst4   (xP10, xM10, xP20,
xM20, Data_in_A[6], Data_in_A[5], ps0[8]);
booth_decoder       inst5   (xP10, xM10, xP20,
xM20, Data_in_A[5], Data_in_A[4], ps0[7]);
booth_decoder       inst6   (xP10, xM10, xP20,
xM20, Data_in_A[4], Data_in_A[3], ps0[6]);
booth_decoder       inst7   (xP10, xM10, xP20,
xM20, Data_in_A[3], Data_in_A[2], ps0[5]);
booth_decoder       inst8   (xP10, xM10, xP20,
xM20, Data_in_A[2], Data_in_A[1], ps0[4]);
booth_decoder       inst9   (xP10, xM10, xP20,
xM20, Data_in_A[1], Data_in_A[0], ps0[3]);
booth_decoder       inst10  (xP10, xM10, xP20,
xM20, Data_in_A[0], 1'b0, ps0[2]);
assign ps0[1]= xM10 | xM20;
assign ps0[10]=~ temp1;

booth_encoder       inst13  (Data_in_B[1],
Data_in_B[2], Data_in_B[3], xP11, xM11, xP21, xM21);

booth_decoder       inst14  (xP11, xM11, xP21,
xM21, Data_in_A[7], Data_in_A[7], temp2);
booth_decoder       inst15  (xP11, xM11, xP21,
xM21, Data_in_A[7], Data_in_A[6], ps1[9]);
booth_decoder       inst16  (xP11, xM11, xP21,
xM21, Data_in_A[6], Data_in_A[5], ps1[8]);
booth_decoder       inst17  (xP11, xM11, xP21,
xM21, Data_in_A[5], Data_in_A[4], ps1[7]);
```

```
booth_decoder      inst18  (xP11, xM11, xP21,
xM21, Data_in_A[4], Data_in_A[3], ps1[6]);
booth_decoder      inst19  (xP11, xM11, xP21,
xM21, Data_in_A[3], Data_in_A[2], ps1[5]);
booth_decoder      inst20  (xP11, xM11, xP21,
xM21, Data_in_A[2], Data_in_A[1], ps1[4]);
booth_decoder      inst21  (xP11, xM11, xP21,
xM21, Data_in_A[1], Data_in_A[0], ps1[3]);
booth_decoder      inst22  (xP11, xM11, xP21,
xM21, Data_in_A[0], 1'b0, ps1[2]);
assign ps1[1] = xM11 | xM21;
assign ps1[10]=~ temp2;

booth_encoder      inst25  (Data_in_B[3],
Data_in_B[4], Data_in_B[5], xP12, xM12, xP22, xM22);

booth_decoder      inst26  (xP12, xM12, xP22,
xM22, Data_in_A[7], Data_in_A[7], temp3);
booth_decoder      inst27  (xP12, xM12, xP22,
xM22, Data_in_A[7], Data_in_A[6], ps2[9]);
booth_decoder      inst28  (xP12, xM12, xP22,
xM22, Data_in_A[6], Data_in_A[5], ps2[8]);
booth_decoder      inst29  (xP12, xM12, xP22,
xM22, Data_in_A[5], Data_in_A[4], ps2[7]);
booth_decoder      inst30  (xP12, xM12, xP22,
xM22, Data_in_A[4], Data_in_A[3], ps2[6]);
booth_decoder      inst31  (xP12, xM12, xP22,
xM22, Data_in_A[3], Data_in_A[2], ps2[5]);
booth_decoder      inst32  (xP12, xM12, xP22,
xM22, Data_in_A[2], Data_in_A[1], ps2[4]);
booth_decoder      inst33  (xP12, xM12, xP22,
xM22, Data_in_A[1], Data_in_A[0], ps2[3]);
booth_decoder      inst34  (xP12, xM12, xP22,
xM22, Data_in_A[0], 1'b0, ps2[2]);
assign ps2[1]= xM12 | xM22;
assign ps2[10]=~ temp3;

booth_encoder      inst37  (Data_in_B[5], Data_in_B[6],
Data_in_B[7], xP13, xM13, xP23, xM23);

booth_decoder      inst38  (xP13, xM13, xP23,
xM23, Data_in_A[7], Data_in_A[7], temp4);
booth_decoder      inst39  (xP13, xM13, xP23,
xM23, Data_in_A[7], Data_in_A[6], ps3[9]);
booth_decoder      inst40  (xP13, xM13, xP23,
xM23, Data_in_A[6], Data_in_A[5], ps3[8]);
booth_decoder      inst41  (xP13, xM13, xP23,
xM23, Data_in_A[5], Data_in_A[4], ps3[7]);
```

```
booth_decoder      inst42  (xP13, xM13, xP23,
xM23, Data_in_A[4], Data_in_A[3], ps3[6]);
booth_decoder      inst43  (xP13, xM13, xP23,
xM23, Data_in_A[3], Data_in_A[2], ps3[5]);
booth_decoder      inst44  (xP13, xM13, xP23,
xM23, Data_in_A[2], Data_in_A[1], ps3[4]);
booth_decoder      inst45  (xP13, xM13, xP23,
xM23, Data_in_A[1], Data_in_A[0], ps3[3]);
booth_decoder      inst46  (xP13, xM13, xP23,
xM23, Data_in_A[0], 1'b0, ps3[2]);
assign ps3[1]= xM13 | xM23;
assign ps3[10]=~ temp4;

half_adder  inst49  (ps1[9], 1'b1, s1[8], c1[8]);
full_adder  inst50  (ps1[8], ps0[10], 1'b1, s1[7], c1[7]);
half_adder  inst51  (ps1[7], ps0[9], s1[6], c1[6]);
half_adder  inst52  (ps1[6], ps0[8], s1[5], c1[5]);
half_adder  inst53  (ps1[5], ps0[7], s1[4], c1[4]);
half_adder  inst54  (ps1[4], ps0[6], s1[3], c1[3]);
half_adder  inst55  (ps1[3], ps0[5], s1[2], c1[2]);
half_adder  inst56  (ps1[2], ps0[4], s1[1], c1[1]);

half_adder  inst57  (ps2[9], 1'b1, s2[8], c2[8]);
full_adder  inst58  (ps2[8], ps1[10], c1[8], s2[7], c2[7]);
full_adder  inst59  (ps2[7], s1[8], c1[7], s2[6], c2[6]);
full_adder  inst60  (ps2[6], s1[7], c1[6], s2[5], c2[5]);
full_adder  inst61  (ps2[5], s1[6], c1[5], s2[4], c2[4]);
full_adder  inst62  (ps2[4], s1[5], c1[4], s2[3], c2[3]);
full_adder  inst63  (ps2[3], s1[4], c1[3], s2[2], c2[2]);
full_adder  inst64  (ps2[2], s1[3], c1[2], s2[1], c2[1]);

half_adder  inst65  (ps3[9], 1'b1, s3[8], c3[8]);
full_adder  inst66  (ps3[8], ps2[10], c2[8], s3[7], c3[7]);
full_adder  inst67  (ps3[7], s2[8], c2[7], s3[6], c3[6]);
full_adder  inst68  (ps3[6], s2[7], c2[6], s3[5], c3[5]);
full_adder  inst69  (ps3[5], s2[6], c2[5], s3[4], c3[4]);
full_adder  inst70  (ps3[4], s2[5], c2[4], s3[3], c3[3]);
full_adder  inst71  (ps3[3], s2[4], c2[3], s3[2], c3[2]);
full_adder  inst72  (ps3[2], s2[3], c2[2], s3[1], c3[1]);
assign A = {1'b0,ps3[10],s3[8],s3[7],s3[6],
s3[5],s3[4],s3[3],s3[2:1],s2[2:1],s1[2:1],ps0[3:2]};
assign B = {1'b1,c3[8],c3[7],c3[6],c3[5],c3[4]
,c3[3],c3[2],c3[1],ps3[1],c2[1],ps2[1],c1[1],
ps1[1],1'b0,ps0[1]};

ripple_carry_adder #(.BIT_WIDTH((2*BIT_WIDTH)))
inst73(A,B,1'b0,Data_out_sum,cout);
endmodule
```

Listing A.2. *4-bit Modified Booth multiplication*

```
/*
* Name: modified_booth_4x4.sv
* Date: 16_05_2022
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description :
*  4*4 modified booth.
*/
module modified_booth_4x4 #(
        parameter int BIT_WIDTH = 4 )
    (
        input logic [BIT_WIDTH-1:0] Data_in_A,Data_in_B,
        output logic [(2*BIT_WIDTH)-1:0] Data_out_sum
    );

    logic       [BIT_WIDTH+1:0] ps0, ps1;
    logic       [BIT_WIDTH-1:0] s1;
    logic       [BIT_WIDTH-1:0] c1;

    booth_encoder       inst1   (1'b0, Data_in_B[0],
    Data_in_B[1], xP10, xM10, xP20, xM20);

    booth_decoder       inst2   (xP10, xM10, xP20,
    xM20, Data_in_A[3], Data_in_A[3], temp1);
    booth_decoder       inst3   (xP10, xM10, xP20,
    xM20, Data_in_A[3], Data_in_A[2], ps0[4]);
    booth_decoder       inst4   (xP10, xM10, xP20,
    xM20, Data_in_A[2], Data_in_A[1], ps0[3]);
    booth_decoder       inst5   (xP10, xM10, xP20,
    xM20, Data_in_A[1], Data_in_A[0], ps0[2]);
    booth_decoder       inst6   (xP10, xM10, xP20,
    xM20, Data_in_A[0], 1'b0, ps0[1]);

    assign ps0[0] = xM10 | xM20;
    assign ps0[5] =~ temp1;

    booth_encoder       inst8   (Data_in_B[1],
    Data_in_B[2], Data_in_B[3], xP11, xM11, xP21, xM21);

    booth_decoder       inst9   (xP11, xM11, xP21,
    xM21, Data_in_A[3], Data_in_A[3], temp2);
    booth_decoder       inst10  (xP11, xM11, xP21,
    xM21, Data_in_A[3], Data_in_A[2], ps1[4]);
    booth_decoder       inst11  (xP11, xM11, xP21,
    xM21, Data_in_A[2], Data_in_A[1], ps1[3]);
    booth_decoder       inst12  (xP11, xM11, xP21,
    xM21, Data_in_A[1], Data_in_A[0], ps1[2]);
    booth_decoder       inst13  (xP11, xM11, xP21,
    xM21, Data_in_A[0], 1'b0, ps1[1]);
    assign ps1[0] = xM11 | xM21;
    assign ps1[5] =~ temp2;

    half_adder  inst14  (ps1[4], 1'b1, s1[3], c1[3]);
    full_adder  inst15  (ps1[3], ps0[5], 1'b1, s1[2], c1[2]);
    half_adder  inst16  (ps1[2], ps0[4], s1[1], c1[1]);
    half_adder  inst17  (ps1[1], ps0[3], s1[0], c1[0]);

    ripple_carry_adder  inst18  ({1'b0,ps1[5],s1[3],s1[2],s1[1:0],ps0[2:1]},
    {1'b1,c1[3],c1[2],c1[1],c1[0],ps1[0],1'b0,ps0[0]}
    ,1'b0,Data_out_sum,rca);


endmodule
```

Listing A.3. *2-bit Modified Booth multiplication*

```
/*
* Name: modified_booth_2x2.sv
* Date: 16_05_2022
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description :
*  2*2 modified booth.
*/
module modified_booth_2x2 #(
        parameter int BIT_WIDTH = 2 )
    (
        input logic  [BIT_WIDTH-1:0] Data_in_A,Data_in_B,
        output logic [(2*BIT_WIDTH)-1:0] Data_out_sum
    );

    logic       [BIT_WIDTH+1:0] ps0;
    logic       [BIT_WIDTH-1:0] s0;
    logic       [BIT_WIDTH-1:0] c0;

    booth_encoder       inst1(1'b0, Data_in_B[0],
    Data_in_B[1], xP10, xM10, xP20, xM20);

    booth_decoder       inst2(xP10, xM10, xP20,
    xM20, Data_in_A[1], Data_in_A[1], temp1);
    booth_decoder       inst3(xP10, xM10, xP20, xM20,
    Data_in_A[1], Data_in_A[0], ps0[2]);
    booth_decoder         inst4(xP10, xM10, xP20, xM20,
    Data_in_A[0], 1'b0, ps0[1]);
    assign ps0[0] = xM10 | xM20;
    assign ps0[3] =~ temp1;

    ripple_carry_adder #(.BIT_WIDTH(4))
    inst7({1'b0,ps0[3],ps0[2:1]},{1'b1,1'b1,1'b0,ps0[0]}
    ,1'b0,Data_out_sum,rca);

endmodule
```

Listing A.4. *Flexible Modified Booth signed multiplication*

```
/*
* Name: flexible_modified_booth_multiplier.sv
* Date: 14_06_2022
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description :
*
*/
module flexible_modified_booth_multiplier #(parameter int BIT_WIDTH = 8 )
    (
        input logic [BIT_WIDTH-1:0] Data_in_A,Data_in_B,
        input logic [1:0] select,
        output logic [(2*BIT_WIDTH)-1:0] Data_out_sum
    );

    logic [(2*BIT_WIDTH)-1:0] Data_out_sum_s_8,Data_out_sum_s_1,Data_out_sum_s_2
    ,Data_out_sum_s;


    // one 8*8 multiplier
    modified_booth #(.BIT_WIDTH(8)) mul_8x8(
        .Data_in_A(Data_in_A),
        .Data_in_B(Data_in_B),
        .Data_out_sum(Data_out_sum_s_8)
    );

    // two 4*4 multiplier          BIT_WIDTH/2
    modified_booth_4x4 #(.BIT_WIDTH(4)) mul_4x4_1(
        .Data_in_A(Data_in_A[3:0]),
        .Data_in_B(Data_in_B[3:0]),
        .Data_out_sum(Data_out_sum_s_1[7:0])
    );
    modified_booth_4x4 #(.BIT_WIDTH(4)) mul_4x4_2(
        .Data_in_A(Data_in_A[7:4]),
        .Data_in_B(Data_in_B[7:4]),
        .Data_out_sum(Data_out_sum_s_1[15:8])
    );

    // four 2*2 multiplier
    // replace it with radix 2 booth multiplier.
    modified_booth_2x2 mul1(Data_in_A[1:0],
    Data_in_B[1:0],
    Data_out_sum_s_2[3:0]);
    modified_booth_2x2 mul2(Data_in_A[3:2],Data_in_B[3:2],
    Data_out_sum_s_2[7:4]);
    modified_booth_2x2 mul3(Data_in_A[5:4],Data_in_B[5:4],
    Data_out_sum_s_2[11:8]);
    modified_booth_2x2 mul4(Data_in_A[7:6],Data_in_B[7:6],
    Data_out_sum_s_2[15:12]);

    mux_4x1 #(.BIT_WIDTH((2*BIT_WIDTH)))
    mux_inst(.a(Data_out_sum_s_2),.b(Data_out_sum_s_8),
    .c(Data_out_sum_s_8),.d(Data_out_sum_s_1),
    .sel(select),.f(Data_out_sum));
    assign Data_out_sum_s = Data_out_sum;
endmodule : flexible_modified_booth_multiplier
```

A.1.2 Vedic multiplier

A.5 shows the signed Vedic multiplier. To calculate the signed multiplication for: Array, Braun and wallace multiplier same code structure is used just the Vedic multiplier is replaced by unsigned version of mentioned multipliers.

Listing A.5. *Example of signed multiplication*

```
/*
* Name: signed_vedic_multiplier.sv
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description : Adding the sign feature.
*/

module signed_vedic_multiplier#(
        parameter int BIT_WIDTH = 8 )(
    input  logic [BIT_WIDTH-1:0] Data_in_A,
    input  logic [BIT_WIDTH-1:0] Data_in_B,
    //input  logic  [1:0] select,
    output logic [(2*BIT_WIDTH-1):0] Data_out_sum
    );
```

```
logic [BIT_WIDTH-1:0] data_a_tc_s;
logic [BIT_WIDTH-1:0] data_b_tc_s;

logic [BIT_WIDTH-1:0] data_a_mux_s;
logic [BIT_WIDTH-1:0] data_b_mux_s;

logic [(2*BIT_WIDTH-1):0] data_out_mul_s;
logic [(2*BIT_WIDTH-1):0] data_out_mul_tc_s;
logic [(2*BIT_WIDTH-1):0] data_out_sum_s;

logic ctrl1 ,ctrl2 ,sel;


twos_complement #(.BIT_WIDTH(BIT_WIDTH))
twos_complement_a (
.Data_in_A(Data_in_A)
,.Data_out_twos_complement(data_a_tc_s));
twos_complement #(.BIT_WIDTH(BIT_WIDTH))
twos_complement_b (
.Data_in_A(Data_in_B),
.Data_out_twos_complement(data_b_tc_s));

/*
signed_signed:      ctrl1: MSB_A * 1 , ctrl2: MSB_B * 1
unsigned_unsigned:  ctrl1: MSB_A * 0 , ctrl2: MSB_B * 0
signed_unsigned:    ctrl1: MSB_A * 1 , ctrl2: MSB_B * 0
unsigned_signed:    ctrl1: MSB_A * 0 , ctrl2: MSB_B * 1
*/
// Here we would like to have signed_signed multiplication
assign ctrl1 = Data_in_A[BIT_WIDTH-1] & 1'b1;
assign ctrl2 = Data_in_B[BIT_WIDTH-1] & 1'b1;

mux_2x1 #(.BIT_WIDTH(BIT_WIDTH)) mux_inst1
(.a(Data_in_A), .b(data_a_tc_s), .sel(ctrl1),.f(data_a_mux_s));
mux_2x1 #(.BIT_WIDTH(BIT_WIDTH)) mux_inst2
(.a(Data_in_B), .b(data_b_tc_s), .sel(ctrl2),.f(data_b_mux_s));

// NxN Unsigned multiplier instantiation
vedic_multiplier #(.BIT_WIDTH(BIT_WIDTH)) i_dut (
    .Data_in_A(data_a_mux_s),
    .Data_in_B(data_b_mux_s),
    .Data_out_sum(data_out_mul_s));

assign sel = ctrl1 ^ ctrl2;

twos_complement #(.BIT_WIDTH(2*BIT_WIDTH))
twos_complement_mul_result(.Data_in_A(data_out_mul_s),
.Data_out_twos_complement(data_out_mul_tc_s));
```

```
    mux_2x1 #(.BIT_WIDTH(2*BIT_WIDTH)) mux_inst3
    (.a(data_out_mul_s), .b(data_out_mul_tc_s), .sel(sel),.f(data_out_sum_s));

    assign Data_out_sum = data_out_sum_s;

endmodule
```

Listing A.6. *8-bit Vedic multiplier*

```
/*
* Name: vedic_multiplier.sv
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description : vedic multiplier 8*8
*
*/
module vedic_multiplier#(
        parameter int BIT_WIDTH = 8 )(
    input  logic [BIT_WIDTH-1:0] Data_in_A,
    input  logic [BIT_WIDTH-1:0] Data_in_B,
    output logic [(2*BIT_WIDTH-1):0] Data_out_sum
    );

    logic [2:0] rca_inst_carry;
    logic [7:0] temp1;
    logic [7:0] temp2;
    logic [7:0] temp3;
    logic [9:0] temp4;
    logic [9:0] temp5;
    logic [7:0] temp6;
    logic [7:0] temp7;

    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_1
    (
        .Data_in_A(Data_in_A[3:0]),
        .Data_in_B(Data_in_B[3:0]),
        .Data_out_sum(temp1)
    );
    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_3
    (
        .Data_in_A(Data_in_A[7:4]),
        .Data_in_B(Data_in_B[3:0]),
        .Data_out_sum(temp2)
    );
    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_2
    (
        .Data_in_A(Data_in_A[3:0]),
        .Data_in_B(Data_in_B[7:4]),
        .Data_out_sum(temp3)
    );
    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_4
    (
        .Data_in_A(Data_in_A[7:4]),
        .Data_in_B(Data_in_B[7:4]),
        .Data_out_sum(temp6)
    );

    ripple_carry_adder #(.BIT_WIDTH(10)) rca_inst1(
        .Data_in_A({2'b00,temp2}),
        .Data_in_B({2'b00,temp3}),
        .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
        .Data_out_Sum(temp4),
        .Data_out_Carry(rca_inst_carry[0])
    );

    ripple_carry_adder #(.BIT_WIDTH(10)) rca_inst2(
        .Data_in_A(temp4),
        .Data_in_B({6'b0000,temp1[7:4]}),
        .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
        .Data_out_Sum(temp5),
        .Data_out_Carry(rca_inst_carry[1])
    );

    ripple_carry_adder #(.BIT_WIDTH(BIT_WIDTH)) rca_inst3(
        .Data_in_A(temp6),
        .Data_in_B({2'b00,temp5[9:4]}),
        .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
        .Data_out_Sum(temp7),
        .Data_out_Carry(rca_inst_carry[2])
    );

    assign Data_out_sum[3:0]  = temp1[3:0];
    assign Data_out_sum[7:4]  = temp5[3:0];
    assign Data_out_sum[15:8] = temp7;

endmodule : vedic_multiplier
```

Listing A.7. *4-bit Vedic multiplier*

```
/*
* Name: vedic_multiplier_4x4.sv
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
```

```
* Description : vedic multiplier 4*4
*/
module vedic_multiplier_4x4#(
        parameter int BIT_WIDTH = 4 )(
    input logic [BIT_WIDTH-1:0] Data_in_A,
    input logic [BIT_WIDTH-1:0] Data_in_B,
    output logic [(2*BIT_WIDTH)-1:0] Data_out_sum
    );
    logic [3:0] temp1;
    logic [3:0] temp2;
    logic [3:0] temp3;
    logic [5:0] temp4;
    logic [5:0] temp5;
    logic [3:0] temp6;
    logic [3:0] temp7;
    logic [5:0] w1;

    //logic c_in = 1'b0;

    logic first_rca_inst_carry;
    logic second_rca_inst_carry;
    logic third_rca_inst_carry;

    vedic_multiplier_2x2 vm_inst_1(
        .Data_in_A(Data_in_A[1:0]),
        .Data_in_B(Data_in_B[1:0]),
        .Data_out_sum(temp1)
    );
    vedic_multiplier_2x2 vm_inst_2(
        .Data_in_A(Data_in_A[3:2]),
        .Data_in_B(Data_in_B[1:0]),
        .Data_out_sum(temp2)
    );
    vedic_multiplier_2x2 vm_inst_3(
        .Data_in_A(Data_in_A[1:0]),
        .Data_in_B(Data_in_B[3:2]),
        .Data_out_sum(temp3)
    );
    vedic_multiplier_2x2 vm_inst_4(
        .Data_in_A(Data_in_A[3:2]),
        .Data_in_B(Data_in_B[3:2]),
        .Data_out_sum(temp6)
    );

    ripple_carry_adder #(.BIT_WIDTH(6)) rca_inst1(
        .Data_in_A({2'b00,temp3}),
        .Data_in_B({2'b00,temp2}),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp4),
        .Data_out_Carry(first_rca_inst_carry)
    );

    assign w1 = {4'b0000,temp1[3:2]};

    ripple_carry_adder #(.BIT_WIDTH(6))  rca_inst2(
        .Data_in_A(temp4),
        .Data_in_B(w1),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp5),
        .Data_out_Carry(second_rca_inst_carry) // It is not connected
    );

    ripple_carry_adder #(.BIT_WIDTH(4)) rca_inst3(
        .Data_in_A(temp6),
        .Data_in_B(temp5[5:2]),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp7),
        .Data_out_Carry(third_rca_inst_carry)
    );

    assign Data_out_sum[1:0] = temp1[1:0];
    assign Data_out_sum[3:2] = temp5[1:0];
    assign Data_out_sum[7:4] = temp7;

endmodule : vedic_multiplier_4x4
```

Listing A.8. *2-bit Vedic multiplier*

```
/*
* Name: vedic_multiplier_2x2.sv
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description : vedic multiplier 2*2
*/
module vedic_multiplier_2x2(
    input logic [1:0] Data_in_A,
    input logic [1:0] Data_in_B,
    output logic [3:0] Data_out_sum
    );

    logic ha1_carry;
    logic [2:0] result_of_and;


    assign Data_out_sum[0]  = Data_in_A[0] & Data_in_B[0];
    assign result_of_and[0] = Data_in_A[1] & Data_in_B[0];
    assign result_of_and[1] = Data_in_A[0] & Data_in_B[1];
    assign result_of_and[2] = Data_in_A[1] & Data_in_B[1];

    half_adder  ha1(
        .Data_in_A(result_of_and[0]),
        .Data_in_B(result_of_and[1]),
        .Data_out_Sum(Data_out_sum[1]),
        .Data_out_Carry(ha1_carry)
    );

    half_adder  ha2(
        .Data_in_A(result_of_and[2]),
        .Data_in_B(ha1_carry),
        .Data_out_Sum(Data_out_sum[2]),
        .Data_out_Carry(Data_out_sum[3])
    );

endmodule : vedic_multiplier_2x2
```

Listing A.9. *Ripple carry adder*

```
/*
* Name: ripple_carry_adder.sv
*
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
* Description : Flexible bit precision Ripple carry adder
*/
module ripple_carry_adder#(
        parameter int BIT_WIDTH = 8 )(
    input logic[BIT_WIDTH-1:0] Data_in_A,
    input logic[BIT_WIDTH-1:0] Data_in_B,
    input logic Data_in_C,
    output logic[BIT_WIDTH-1:0] Data_out_Sum,
    output logic Data_out_Carry);

    wire [BIT_WIDTH-1:0] sum;
    wire [BIT_WIDTH-1:0] carry;

    full_adder fa1(
        .Data_in_A(Data_in_A[0]),
        .Data_in_B(Data_in_B[0]),
        .Data_in_C(Data_in_C),
        .Data_out_Sum (sum[0]),
        .Data_out_Carry (carry[0])
    );
    genvar  i;
    generate
        for (i=1; i<BIT_WIDTH; i++)
        begin : generate_full_adder_inst
            full_adder full_adder_inst
                (
                    .Data_in_A(Data_in_A[i]),
                    .Data_in_B(Data_in_B[i]),
                    .Data_in_C(carry[i-1]),
                    .Data_out_Sum(sum[i]),
                    .Data_out_Carry(carry[i])
                );
        end
    endgenerate

    assign Data_out_Sum= sum;
    assign Data_out_Carry = carry[BIT_WIDTH-1];

endmodule : ripple_carry_adder
```

Listing A.10. *Flexible Vedic multiplier*

```
/*
* Name: vedic_mul_flex.sv
* Author: Samaneh Ammari
* Copyright (C) 2022 Tampere university
*
* Description : Flexible Vedic multiplier
* Flow_mul_flex
*/
module vedic_mul_flex#(
        parameter int BIT_WIDTH = 8 )(
    input  logic [1:0] sel,
    input  logic [BIT_WIDTH-1:0] Data_in_A,
    input  logic [BIT_WIDTH-1:0] Data_in_B,
    output logic [(2*BIT_WIDTH-1):0] Data_out_sum
    );
    logic [(2*BIT_WIDTH-1):0] Data_out_sum_4,Data_out_sum_8,Data_out_sum_2;
    logic [2:0] rca_inst_carry;
    logic [7:0] temp1;
    logic [7:0] temp2;
    logic [7:0] temp3;
    logic [9:0] temp4;
    logic [9:0] temp5;
    logic [7:0] temp6;
    logic [7:0] temp7;

    logic [3:0] temp1_2x2;
    logic [3:0] temp2_2x2;
    logic [3:0] temp3_2x2;
    logic [5:0] temp4_2x2;
    logic [5:0] temp5_2x2;
    logic [3:0] temp6_2x2;
    logic [3:0] temp7_2x2;
    logic [5:0] w1_2x2;

    logic first_rca_inst_carry;
    logic second_rca_inst_carry;
    logic third_rca_inst_carry;

    vedic_multiplier_2x2 vm_inst_1(
        .Data_in_A(Data_in_A[1:0]),
        .Data_in_B(Data_in_B[1:0]),
        .Data_out_sum(temp1_2x2)
    );
    vedic_multiplier_2x2 vm_inst_2(
        .Data_in_A(Data_in_A[3:2]),
        .Data_in_B(Data_in_B[1:0]),
        .Data_out_sum(temp2_2x2)
    );
    vedic_multiplier_2x2 vm_inst_3(
        .Data_in_A(Data_in_A[1:0]),
        .Data_in_B(Data_in_B[3:2]),
        .Data_out_sum(temp3_2x2)
    );
    vedic_multiplier_2x2 vm_inst_4(
        .Data_in_A(Data_in_A[3:2]),
        .Data_in_B(Data_in_B[3:2]),
        .Data_out_sum(temp6_2x2)
    );
    assign Data_out_sum_2 = {temp6_2x2,temp1_2x2};

    ripple_carry_adder #(.BIT_WIDTH(6)) rca_inst1_2x2(
        .Data_in_A({2'b00,temp3_2x2}),
        .Data_in_B({2'b00,temp2_2x2}),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp4_2x2),
        .Data_out_Carry(first_rca_inst_carry)
    );

    //assign result_vm_inst_4 = {first_rca_inst_carry,rca_inst_inst_sum};
    assign w1_2x2 = {4'b0000,temp1_2x2[3:2]};

    ripple_carry_adder #(.BIT_WIDTH(6))  rca_inst2_2x2(
        .Data_in_A(temp4_2x2),
        .Data_in_B(w1_2x2),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp5_2x2),
        .Data_out_Carry(second_rca_inst_carry) // It is not connected
    );

    ripple_carry_adder #(.BIT_WIDTH(4)) rca_inst3_2x2(
        .Data_in_A(temp6_2x2),
        .Data_in_B(temp5_2x2[5:2]),
        .Data_in_C(1'b0),
        .Data_out_Sum(temp7_2x2),
        .Data_out_Carry(third_rca_inst_carry)
    );

    assign temp1[1:0] = temp1_2x2[1:0];
    assign temp1[3:2] = temp5_2x2[1:0];
    assign temp1[7:4] = temp7_2x2;


    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_3
    (
        .Data_in_A(Data_in_A[7:4]),
        .Data_in_B(Data_in_B[3:0]),
        .Data_out_sum(temp2)
    );
    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_2
    (
        .Data_in_A(Data_in_A[3:0]),
        .Data_in_B(Data_in_B[7:4]),
        .Data_out_sum(temp3)
    );
    vedic_multiplier_4x4 #(.BIT_WIDTH(4)) vm_4b_inst_4
    (
        .Data_in_A(Data_in_A[7:4]),
        .Data_in_B(Data_in_B[7:4]),
        .Data_out_sum(temp6)
    );
    ripple_carry_adder #(.BIT_WIDTH(10)) rca_inst1(
    .Data_in_A({2'b00,temp2}),
    .Data_in_B({2'b00,temp3}),
    .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
    .Data_out_Sum(temp4),
    .Data_out_Carry(rca_inst_carry[0])
    );

    ripple_carry_adder #(.BIT_WIDTH(10)) rca_inst2(
    .Data_in_A(temp4),
    .Data_in_B({6'b0000,temp1[7:4]}),
    .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
```

```
        .Data_out_Sum(temp5),
        .Data_out_Carry(rca_inst_carry[1])
        );

        ripple_carry_adder #(.BIT_WIDTH(BIT_WIDTH)) rca_inst3(
        .Data_in_A(temp6),
        .Data_in_B({2'b00,temp5[9:4]}),
        .Data_in_C(1'b0), // It is not connected :carry_rca_inst1
        .Data_out_Sum(temp7),
        .Data_out_Carry(rca_inst_carry[2])
        );

        assign Data_out_sum_4 = {temp2,temp1};
        assign Data_out_sum_8 = {temp7,temp5[3:0],temp1[3:0]};

        mux_4x1 #(.BIT_WIDTH((2*BIT_WIDTH)))
        mux_inst(.a(Data_out_sum_2),.b(Data_out_sum_8)
        ,.c(Data_out_sum_8),.d(Data_out_sum_4),.sel(sel),
        .f(Data_out_sum));

endmodule : vedic_mul_flex
```