# Design of Efficient DNN Accelerator Architectures

A thesis submitted to the
College of Graduate and Postdoctoral Studies (CGPS)
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Electrical & Computer Engineering
University of Saskatchewan
Saskatoon, Canada

By

Mohammadreza Asadikouhanjani

# Permission to Use

In presenting this thesis/dissertation in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis/dissertation in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis/dissertation work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis/dissertation or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis/dissertation.

# Disclaimer

Reference in this thesis to any specific commercial products,process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes

Request for permission to copy or to make any other use of material in this thesis in whole or in part should be addressed to:

Head of the Division of Biomedical Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

# Abstract

Deep Neural Networks (DNNs) are the fundamental processing unit behind modern Artificial Intelligence (AI). Accordingly, expecting a future with smart devices that are able to monitor, decide, and take action seems reasonable. However, DNNs are computation and power-hungry, which makes deployment of them into edge devices challenging. The focus of this dissertation is on designing architectures to perform the inference of DNNs efficiently. The contents of this dissertation can be divided into four specific areas: (1) early detection of the ineffectual computations inside the computation engine; (2) enhancing the utilization of Processing Elements (PEs) inside the computation engine; (3) skipping identical effectual computations through binary Multiply and Accumulation (MAC) operations; (4) the design of approximate DNN accelerators.

In most DNNs, an activation function follows a convolutional or a fully connected layer. Several popular activation functions involve setting all negative inputs to zero. In this dissertation, firstly, the characteristics of activation layers that are considered for adding non-linearity to DNNs are studied. Then, a novel architecture in which the activation function is merged with the prior computational layer is proposed. To add more detail, the proposed architecture coordinates early sign detection of output features. When compared to the original design, our method achieves a speedup of $\times$2.19 and reduces energy consumption by $\times$1.94. The average reduction in the number of multiply-accumulate (MAC) operations is 10.64% and the average reduction in the number of the load operations is 3.86%. These improvements are achieved while maintaining classification accuracy in two popular benchmark networks.

One of the main challenges that DNN accelerator developers face is keeping all the PEs busy with performing effectual computations while running DNNs. In this dissertation, a Twin-PE for spatial DNN accelerators is introduced that increases the utilization of the PEs and the performance of the whole computation engine. In more detail, the proposed architecture which comes with a negligible area overhead is implemented based on sharing the scratchpads between the PEs to use the available slack time caused by applying computation-pruning techniques. When compared to the reference design, our proposed method achieves a speedup of $\times$1.24 and an energy efficiency of $\times$1.18 per inference.

Decomposing the MAC operations down to bit-level provides the chance of skipping bit-wise and word-wise sparsity. However, there is still room for pruning the effectual computations without reducing the accuracy of DNNs. In this dissertation, a novel real-time architecture by decomposing multiplications down to bit level and pruning identical computations while running benchmark networks. Our proposed design achieves an average per layer speedup of $\times 1.4$ and an energy efficiency of $\times 1.21$ per inference while maintaining the accuracy of benchmark networks.

Applying approximate computing techniques reduces the cost of the underlying circuits so that DNN inference would be performed more efficiently. However, applying approximation to DNNs is somehow different from other applications. In this dissertation, a step-wise approach for implementing a re-configurable Booth multiplier suitable for inference of DNNs is proposed. In addition, the tolerance of different layers of DNNs to approximation is evaluated and the effect of applying various degrees of approximation on inference accuracy is explored. The proposed design achieves an area efficiency of $\times 1.19$ and energy efficiency of $\times 1.28$ compared to the exact design while running benchmark DNNs.

# Acknowledgments

I would like to express my sincere appreciation to my supervisor Dr. Seok-Bum Ko for his continuous support throughout the course of my Ph.D. studies. His immense knowledge and experience in the field has given him an excellent eye for worthwhile research opportunities, and I can not thank him enough for the patience, compassion, and encouragement he has shown me. Without his guidance, this dissertation would not have been possible.

I would also like to express sincere gratitude to my lab-mate Hao Zhang for his vast technical knowledge and the prompt assistance he always gave me when troubleshooting issues in the lab. My lab-mates have continuously taken the extra step to help me out when I need it, and I can not overstate how much I appreciate their support.

I thank my mom, my dad, my sister, and my brothers for their endless patience throughout my studies. It goes without saying that research can impose a sense of intimidation and frustration from time to time. I am so incredibly grateful for the support provided by my family and friends during those times that I was feeling under confident. It was because of their encouraging words that I was also able to experience the joy and accomplishment of seeing my research through.

Last but not the least, my sincere thanks to the Natural Sciences and Engineering Research Council of Canada (NSERC), the R&D program of the Korean Ministry of Trade, Industry and Energy (MOTIE), the Korea Evaluation Institute of Industrial Technology (KEIT), and the Department of Electrical and Computer Engineering, University of Saskatchewan.

To all those who are patient and persistent in encouraging me, you have shown me that I am much more capable than I thought myself to be.

# Table of Contents

# List of Abbreviations

| | |
|---|---|
| ADC | Analog to Digital Converters |
| AI | Artificial Intelligence |
| ASIC | Application-Specific Integrated Circuits |
| CPU | Central Processing Unit |
| DAC | Digital to Analog Converters |
| DNN | Deep Neural Network |
| DRAM | Dynamic Random-Access Memory |
| FFNNs | Feed Forward Neural Networks |
| FPGA | Field Programmable Gate Array |
| GANs | Generative Adversarial Networks |
| GB | Global Buffer |
| GPU | Graphical Processing Unit |
| Ifmap | Input feature map |
| LUT | Look-up Table |
| MAC | Multiply and Accumulate |
| MLP | multi-layer perceptron |
| MXU | Matrix Multiply Unit |
| NFU | Neural Functional Unit |
| nlp | natural language processing |

| | |
|---|---|
| NN | Neural Network |
| NoC | Network on Chip |
| PDP | Power Delay Product |
| PE | Processing Element |
| Psum | Partial sum |
| ReLU | Rectified Linear Unit |
| RNNs | Recurrent Neural Networks |
| SIMD | Single Instruction Multiple Data |
| VPU | Vector Processing Unit |

# List of Tables

# List of Figures

# 1. Introduction

This chapter introduces the research works proposed in this dissertation. This chapter presents the popularity of deep learning in many fields of applications. Due to the revolutionary improvements that come with Artificial Intelligence (AI) based applications that are equipped with a Deep Neural Network (DNN), optimizing hardware deep learning operations becomes necessary. This motivates the research works to be presented in this dissertation that is to design computation engines based on the requirements of the deep learning computation. Section 1.1 presents the overview of the computation complexity of DNNs. The motivation of the proposed works is presented in Section 1.2. Section 1.3 provides an overview of the research works in this dissertation. Section 1.4 presents the contributions of the proposed works. Finally, Section 1.5 lists the publications and submissions made during the course of the Ph.D. study.

## 1.1 Overview of DNNs

Modern Artificial Intelligence (AI) applications are equipped with DNNs that caused a revolutionary improvement in those applications. With the new advancement in the DNNs, the performance of the applications such as speech recognition [10], environment sound classification [11], object detection [12], and self-driving cars [13] is increasing as well. Although DNNs are introduced 60 years ago, it has been roughly a decade that big data are available, and processing the big data in a reasonable amount of time is possible. A big portion of the past decade was spent on increasing the accuracy and performance of DNNs. Accuracy improvement of DNNs comes with a significant computation complexity. It takes 22.6G multiply and accumulate (MAC) operations to perform one inference using ResNet-152 [14] and 46.04G MAC operations to perform one inference using VGG16 [15]. Using an Intel Core i7 CPU, it takes roughly 800 ms to perform

one inference while running VGG16 [15]. The input size of VGG16 is $224 \times 224$. A self driving car that includes 8 $(1920 \times 1280)$ pixels cameras (30 frames per second), requires hardware platform to perform $8 \times (46.06(GigaMAC) \div (800)) \times 30 \times ((1920 \times 1280) \div (224 \times 224)) = 676$ Tera MAC operations per second. Using multiple DNNs enlarges the computation complexity even more.

Researchers have been required to use Graphical Processor Units (GPUs) and Central Processing Units (CPUs) platforms to develop and run DNNs at the beginning of the past decade. The price, large power consumption, and the large latency of such platforms have been a critical challenge for the deployment of the advanced DNNs into edge-devices. Accordingly, researchers have been pushed toward Application-Specific Integrated Circuits (ASICs) for hardware implementation.

Although ASIC platforms are optimized in terms of price and power consumption compared to GPU and CPUs. Many of the AI solutions are battery-dependent, which makes the implementation of DNN accelerators challenging. Advanced DNN models need large Global buffers (GBs). Accessing GBs is 6× more expensive than performing a single 16 bit MAC operation in 65nm technology [1]. in terms of energy consumption. Large DNNs even need DRAM. Accessing DRAM is 200× more expensive than performing a single 16-bit MAC operation in 65nm technology in terms of energy consumption.

### 1.1.1 The Basics

Deep learning is a subcategory of machine learning that itself is a subcategory of AI. Neural Networks (NNs) and their connection are inspired by the human brain. It turns out that adding more layers improves the accuracy of these networks significantly. Technically, the NNs that include hidden layers are called DNNs. Figure 1.1 shows the internal architecture of DNNs. In more detail, the idea of DNNs is to use many layers to hierarchically extract meaningful information from the input data. This transformation, called inference, involves multiple stages, each of which is often referred to as a layer. A DNN often has dozens to hundreds of layers.

In [16], deep learning is defined as representation learning. In other words, deep learning is the automated formation of useful representations from data. In addition, as explained in [16], at a high-level, neural networks could be encoders, decoders, or a combination of both. To give

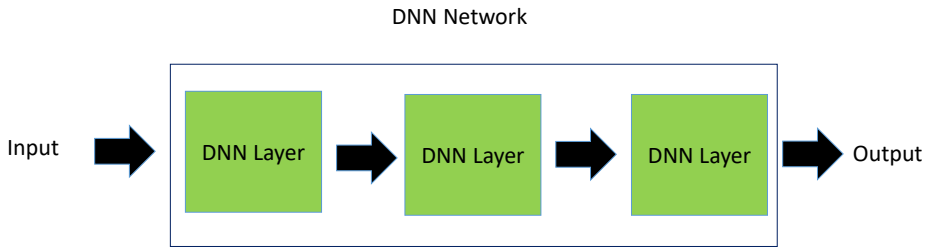


Figure 1.1: Internal architecture of a Deep Neural Network

more detail, encoders find patterns in raw data to form compact, useful representations. However, decoders generate high-resolution data from those representations and the generated data is either new examples or descriptive knowledge.

Within AI and machine learning, there are three basic approaches. One is supervised learning, the second one is unsupervised learning and the last one is reinforcement learning. The main difference between the first two approaches is that supervised learning uses labeled data to help predict classes, while unsupervised learning does not. In what follows these approaches are in more detail.

**What is supervised learning?**

Supervised learning is a subcategory of machine learning which is defined by its use of labeled datasets. These datasets are designed to train or supervise algorithms into classifying data with high accuracy. By using labeled inputs and outputs, the developed model can measure its accuracy and learn over time.

**Feed Forward Neural Networks (FFNNs)**

A network wherein all its connections between the nodes do not form a cycle, is called a feed forward neural network. The history of these types of networks goes back to 1940s. These types of networks also count as the first and simplest type of artificial neural network proposed. These types of networks are also called deep networks, multi-layer perceptron (MLP), or simply neural networks. When input data goes down through the network, each layer processes a specific aspect of the data. Feed forward neural networks are made up of the following:

- **Input layer:** This layer consists of the neurons that receive inputs and pass them on to the other layers.
- **Output layer:** This layer is the predicted feature. Depending on the application, the output could be in continuous or discrete format.
- **Intermediate layer:** These types of layers that usually come with a vast number of neurons are referred to all the layers between input layer and the output layer.
- **Neuron weights:** This term refers to the amplitude of a connection between two neurons.

In what follows, some other types of networks which are modified feed forward neural networks are introduced. Accordingly, many of the fundamental concepts are identical.

**Convolutional Neural Networks (CNNs)**

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. The fundamental layer in these networks is convolutional layer. In more detail, each convolution layer convolves the input and each filter and passes the results to the next layer. One of the most common types of convolution that is used in DNNs is 2-D convolution. In a 2-D convolution, 2-D filters or kernels, slide over a 2-D input data. The 2-D input data is also called 2-D feature map. For every location of the input that each filter slides over, the summation of the element-wise multiplication of the filter weights and the input feature maps (ifmaps) will be computed. Accordingly, the output of a 2-D convolution would be a 2-D output feature which is also called output feature map. Figure 1.2 shows a simple 2-D convolution example. In this figure, I refers to a 2D input feature map and K refers to a 2D filter channel. The intermediate values generated by the many 2-D convolutions for each output activation called partial sums (psums). When the size of the filter and the input data is identical, that layer is called a fully connected layer. Fully connected layers usually are used as the last layers of advanced DNNs.

Each 2-D filter is called a channel. The same thing is true with each 2-D feature map. In practice, the number of channels of each filter and the input data is way more than one (e.g., up to

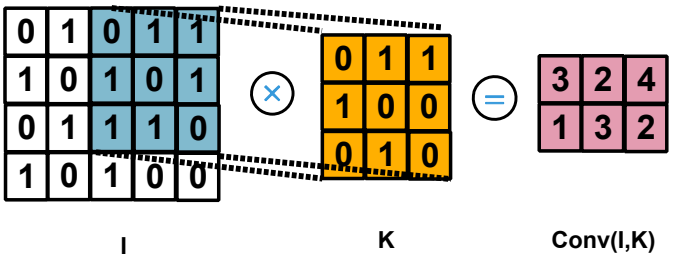


Figure 1.2: A simple 2D convolution example

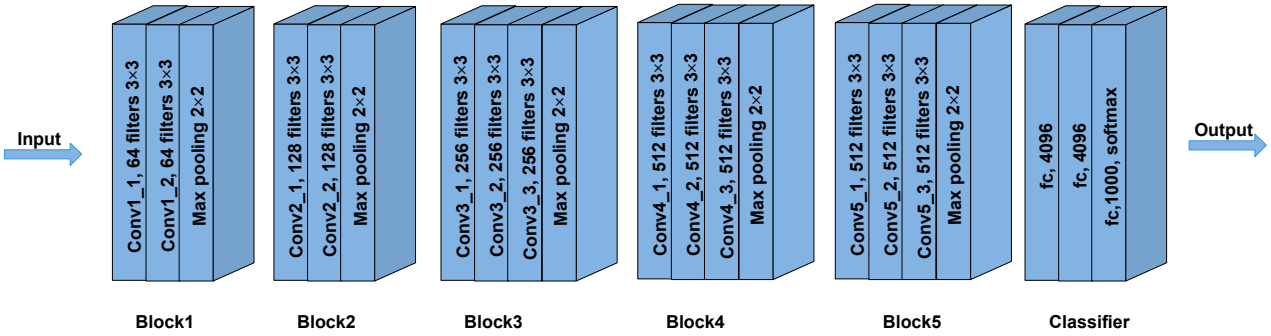


Figure 1.3: Overview of the architecture of VGG16

512 in VGG16). The overview of the architecture of VGG16 is shown in Figure 1.3. Accordingly, it is possible to show each filter and each input data as a 3D matrix. Furthermore, the number of filters of each layer is way more than one (e.g., up to 512 in VGG16). Each channel of the output feature map is the result of the convolution of each filter and its relevant channels and the input feature map and its relevant channel.

Usually, each convolution layer is followed by an activation and a pooling layer in DNNs. The activation layer also known as the activation function is considered to add non-linearity to the network. The activation functions are at the very core of Deep Learning. They determine the output of a model, its accuracy, and computational efficiency. In some cases, activation functions have a major effect on the model's ability to converge and the convergence speed while training. There are different activation functions such as Sigmond or Rectified Linear Unit (ReLU). The ReLU layer, which is very common in most well-known DNNs involves setting all the negative values to zero. The overview of this activation function is shown in Figure 1.4. A pooling layer is another building block of a CNN. Its function is to progressively reduce the spatial size of the

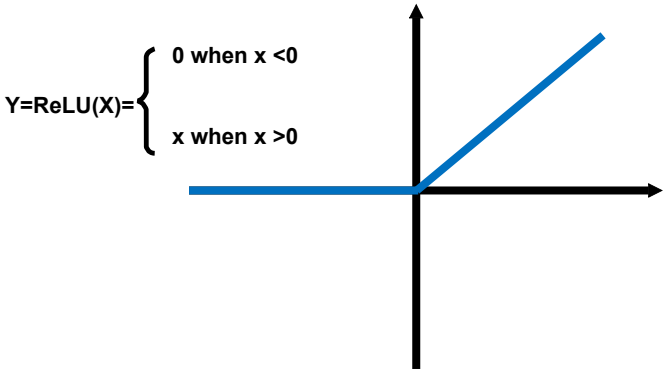


Figure 1.4: Overview of the ReLU activation function

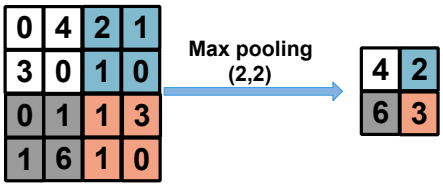


Figure 1.5: A $2 \times 2$ Max pooling example

representation to reduce the number of parameters and computation in the network.

The most common approach used in pooling is Max pooling. Figure 1.5 shows $2 \times 2$ Max pooling using an example. As shown in this figure, a $4 \times 4$ matrix is considered as the input. The pooling operation involves sliding a two-dimensional filter over the input matrix. Considering a stride size of 2 for the given example, there are only 4 regions that the filter runs over the input matrix. These regions are distinguished with different colors in Figure 1.5. When performing Max pooling operation for the given input, the maximum of each of the 4 distinguished regions would be selected and considered as an element of the output matrix. Therefore, for the given example, the output matrix has 4 elements.

**A Recurrent Neural Network (RNN)**

In some application such as language translation, memory is required to process the data. So the main question is that how a feed forward neural network would able to use previous information to affect later ones? A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time-series data. The architecture of these types of networks is shown

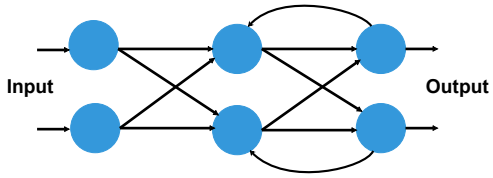


Figure 1.6: The internal architecture of a simple Recurrent Neural Network

in Figure 1.6. As shown in Figure 1.6, there are connections with both forward and backward directions. There are more applications that this architectural paradigm is commonly used for, such as language translation, natural language processing (nlp), speech recognition, and image captioning.

**What is unsupervised learning?**

The unsupervised learning term refers to machine learning approaches that try discover hidden patterns and cluster unlabeled datasets. The data clustering, association, and dimensionality reduction are the main applications of unsupervised learning.

**What is reinforcement learning?**

Reinforcement learning algorithms define how agents should take actions in an environment in order to get rewards or maximize the number of rewards. The agent will get positive feedback for good actions and it will get a penalty for bad actions. To add more detail, reinforcement learning algorithms can make sequential decisions and learn from their experience.

## 1.2 Motivations of Research Works

### 1.2.1 Ineffectual MAC Operations in DNNs

As mentioned in the previous sections, DNNs have enhanced the accuracy of many AI applications unprecedentedly. However, DNNs require hundreds of megabytes of storage and involve high computational complexity. There are many DNN accelerators available implemented on ASIC and field-programmable gate array (FPGA) platforms [17, 18] which are better optimized in terms of power consumption compared to GPU architectures [19].

To reduce the number of MAC operations in DNNs, [20] showed that neighboring output feature map elements tend to have similar values and leveraged this to reduce the required MAC operations by 30.4% with a 1.7% accuracy loss. Authors in [21] equipped processing units with an approximate processing mode and achieved improvements in energy efficiency of 34.11% to 51.47% with an accuracy drop of 5%. In [22], an accelerator is proposed which performs early detection of negative output features via the use of the inverse two's complement format of the filter elements. The design presented in [22] reduces the number of required MAC operations, however, the proposed approach comes with several weak points. For example, by using the proposed architecture in [22], the speedup decreases for the high depth layers since these layers come with more channels. In more detail, in the proposed method in [22], the computations start only when all the relevant input feature maps and filter elements are loaded into the PEs. Furthermore, no results were included in [22] in regards to the first fully connected layer of the benchmark network since it is not possible to map this layer into the proposed accelerator.

### 1.2.2 Divergence of the Workloads in Spatial Designs

Designing an efficient computation engine enhances the performance of these platforms while running DNNs. Energy-efficient DNN accelerators use skipping sparsity and early negative output feature detection to prune the computations. Authors in [23] introduced NoC-based DNN accelerator. They showed that efficient PE and NoC designs are both critically important to keep the system performance high in the NoC-based DNN accelerators. Compared with the conventional DNN accelerators, the number of memory accesses in NoC-based design is reduced by 88% up to

99%.

Eyeriss V2 [24] is a state-of-the-art energy-efficient DNN accelerator that exploits a flexible NoC which provides the required bandwidth of the computation engine while running various DNNs. With the use of the proposed NoC types in [23] and [24] and data reuse techniques, it is possible to decrease the NoC based stalls in sparse DNNs. However, these accelerators perform non-effectual computations for negative outputs. Furthermore, even by providing such NoC for the PEs, applying computation-pruning techniques still causes stalls inside spatial designs. Spatial DNN accelerators in principle can support computation-pruning techniques compared to other common architectures such as systolic arrays. These accelerators need a separate data distribution fabric like buses or trees with support for high bandwidth to run the mentioned techniques efficiently and avoid Network on Chip (NoC) based stalls. Spatial designs suffer from divergence and unequal work distribution. Therefore, applying computation-pruning techniques into a spatial design, which is even equipped with an NoC that supports high bandwidth for the PEs, still causes stalls inside the computation engine.

### 1.2.3 Skipping Identical Bit-wise Computations

Some efficient DNN accelerators apply techniques that target sparsity. Zero computations are considered as ineffectual MAC operations which can be safely eliminated, saving energy and/or storage and communication [1, 25, 26]. In [27], the authors presented a method for transparently identifying ineffectual computations during inference with deep learning models. Specifically, by decomposing multiplications down to the bit level. This work targets bit sparsity, that is zero bits since processing zero bits in a MAC operation does not affect the outcome. As shown in [27], the number of multiplications during inference can be potentially reduced by at least 40× across a wide selection of neural networks. Furthermore, the authors in [27] proposed a novel PE that is 5× smaller and much more energy-efficient than the most competitive works [3]. The proposed architecture in [28] only skips computation cycles for zero-valued weights and the proposed architecture in [29] only skips computation cycles for zero-valued ifmaps. However, in these types of accelerators, the focus is on skipping the ineffectual computations [30, 31].

### 1.2.4 Applying Approximation into DNNs

One of the promising solutions to perform inference of DNNs efficiently is utilizing approximate multipliers inside the computation engine of DNN accelerators [32]. The previous researches have reported that DNNs are resilient against small arithmetic errors [33]. Authors in [32] showed that one of the promising solutions to perform inference of DNNs efficiently is utilizing approximate multipliers inside the computation engine of DNN accelerators. In [34], authors have introduced approximate multipliers based on alphabet-set multiplication. In more detail, the weights are divided into parts, having 4 bits. Multiplication by each 4-bit part of the weight is implemented by shifting a pre-computed input value and followed by summation.

## 1.3 Overview of Research Works

In this dissertation, based on the computation characteristics discussed in Section 1.2, several computation engines optimized for deep learning computation are proposed. We provide the solutions both to optimize the state-of-the-art accelerators for deep learning computation and to design deep learning specific computation units. The computation engines proposed in this dissertation support running different DNN architectures out there. This is different from the design of an architecture that is optimized for a specific model or operation. Therefore, the proposed computation engines are more useful in both ASIC and FPGA based processor designs that are used in servers and datacenters. The whole dissertation is composed of seven chapters explained as follows:

**Chapter 1 Introduction:** covers the basics of different architectural paradigms of machine learning approaches and then describes the motivation behind the research presented in this dissertation, the contributions provided in this dissertation, the publications made during over the course of the Ph.D. study, and the organization of this dissertation.

**Chapter 2 Review of Contemporary DNN Accelerators:** provides a review of contemporary works in the literature which features a broad range of architectures. The efficient architectures reviewed include efficient data-flow, pruning, quantization, skipping sparsity, near memory computation, and approximate MAC units.

**Chapter 3 Early Detection of Negative Output Features:** introduces a novel architecture for reordering the computations in the computation engines. Among the previously proposed architectures that tried to reduce the number of MAC operations while performing inference of DNNs, some come with accuracy loss, which is not always acceptable. In more detail, in some applications such as safety-critical applications, the accuracy loss beyond a certain threshold, while performing inference of DNNs is not acceptable [35]. Some other works maintain the accuracy. However, their considered dataflow is designed poorly that can not coordinate to run all the DNN layers efficiently such as high depth layers [22]. In addition, some other proposed algorithms recommend reordering the computations using indexes, without considering the extra high-level memory access needed for loading those indexes from DRAM, and also the extra needed storage for

keeping those indexes inside the computation engine. Therefore, although efficient early detection of output features come with benefits like less number of MAC operations and less number of accesses to memory, but still there are some challenges that need to be resolved to equip DNN accelerators out there with the discussed technique.

**Chapter 4 Enhancing the Utilization of PEs in Spatial DNN Accelerators** focuses on increasing the utilization of PEs in spatial computation engines. In a spatial architecture, the PEs that perform their tasks earlier have a slack time compared to others. Spatial DNN accelerators come with an array of PEs that are considered for performing the MAC operations of DNNs. Therefore, due to the divergence of workloads, each specific PE might finish its task sooner or later than other PEs. In other words, the PEs that finish their given task earlier have a slack time. However, to avoid designing a complex control unit and also utilizing the chance of local data reuse while distributing the ifmaps and filter weights among the PEs, the main control unit waits until all the PEs are done with their given tasks, then loads the PEs with new tasks. Accordingly, there is a need to assess the conventional architectures out there to evaluate the chance of using the available slack times inside the spatial computation engines to balance the divergence of the workload inside the whole computation engine.

**Chapter 5 Pruning the Effectual Computations in DNNs:** optimizes the available bit-level computation engines. Although skipping sparsity bit-wise and word-wise comes with both energy efficiency and speedup while running different layers of DNNs especially high depth layers. There is still room to skip effectual identical computations while decomposing the ifmaps and filter weights down to bit-level. In more detail, as authors in [20] showed, neighboring output feature map elements tend to have similar values. In addition, the wights of each layer have a Gaussian distribution around zero. To add more information, there is a high chance that the high significant bits in close values to be identical.

**Chapter 6 Efficient Approximate DNN Accelerators for Edge Device:** explores applying approximation into the computation engine of DNN accelerators. There are many approximate multipliers proposed for various applications until now. However, the function of only a few of these approximate designs has been explored while performing inference of DNNs. Furthermore, little attention has been given to applying various approximation techniques to different layers

of DNNs. Lastly, for most of the current approximate neural network implementation, single approximate arithmetic is used for the whole model process. However, a single type of approximate arithmetic unit may not be an optimal solution for DNNs.

**Chapter 7 Conclusions and Future Work:** provides a summary of this dissertation and discusses potential future works.

## 1.4 Contributions of This Dissertation

By considering all the requirements mentioned in Section 1.2, efficient DNN accelerator architectures are proposed in this dissertation. In what follows, the contributions of the proposed accelerators are provided in this dissertation:

- An area-efficient architecture for the early detection of negative output features in hardware while maintaining the accuracy of DNNs is proposed.
- A novel dataflow is introduced that can map various fully connected and convolutional layers and keeps the computation engine busy with effectual computations while reordering the computations.
- A novel real-time architecture for the spatial DNN accelerators which uses the slack time caused by computation-pruning techniques and the used NoC format is developed and explored.
- A novel real-time architecture for spatial DNN accelerators is proposed which not only skips the ineffectual computations but also it prunes the effectual computations during inference while maintaining the accuracy.
- A novel dataflow is proposed for the layers with a limited chance of reuse of the ifmaps and filter weights such as fully connected and depth-wise convolutional layers.
- A detailed step-wise approach for designing a re-configurable approximate Booth multiplier is proposed. Accordingly, it is possible to re-configure the multiplier dynamically at run-time.
- The effect of applying various degrees of approximation to different layers of DNNs is evaluated.

## 1.5 Publications During Ph.D. Research

### 1.5.1 Published Journal Papers

- **M. Asadikouhanjani** and S. -B. Ko, "A Novel Architecture for Early Detection of Negative Output Features in Deep Neural Network Accelerators," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3332-3336, Dec. 2020, doi: 10.1109/TCSII.2020.2977015.

  A major portion of this paper is included in Chapter 3 : Early Detection of Negative Output Features.

- **M. Asadikouhanjani** and S. -B. Ko, "Enhancing the Utilization of Processing Elements in Spatial Deep Neural Network Accelerators," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1947-1951, Sept. 2021, doi: 10.1109/TCAD.2020.3031240.

  A major portion of this paper is included in Chapter 4: Enhancing the Utilization of PEs in Spatial DNN Accelerators.

- **M. Asadikouhanjani**, H. Zhang, L. Gopalakrishnan, H.-J. Lee and S.-B. Ko, "A Real-Time Architecture for Pruning the Effectual Computations in Deep Neural Networks," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 2030-2041, May 2021, doi: 10.1109/TCSI.2021.3060945.

  A major portion of this paper is included in Chapter 5 : Pruning the Effectual Computations in DNNs.

### 1.5.2 Other publications

- **M. Asadikouhanjani**, H. Zhang, and S.-B. Ko, "Efficient Approximate DNN Accelerators for Edge Devices: An Experimental Study," Submitted as a book chapter to Springer (to appear by August 30th, 2022).

  A major portion of this paper is included in Chapter 6: Efficient Approximate DNN Accelerators for Edge Device

### 1.5.3 Other publications not included in this dissertation

- H. Zhang, **M. Asadikouhanjani**, and S.-B. Ko, "Approximate Computing for Efficient Neural Network Computation: A Survey," Submitted as a book chapter to Springer (to appear by August 30th, 2022).

# 2. Review of Contemporary DNN Accelerators

Section 2.1 introduces and compares spatial and temporal architectures. Section 2.2 introduces contemporary state-of-the-art DNN architectures out there.

## 2.1 Spatial Designs vs Temporal Designs

There are different categories considered for classifying the proposed DNN accelerators until now. One classification is in terms of general architecture considered for designing the computation engine of DNN accelerators. In more detail, depending on whether the internal architecture of PEs inside the computation engine contains a specific local control unit and register file or not, it is possible to classify a DNN accelerator as a spatial or temporal architecture. The overview of these architectures is shown in Figure 2.1.

In spatial designs, each Processing Element (PE) has its own control unit and register files. However, a temporal architecture includes one shared large register file and one shared control unit. Spatial architectures are efficient since smaller register files inside the PEs come with less static power consumption and shorter data travel path. However, the internal architecture of each PE is more complex since they have their own control unit. The control unit in each PE coordinates performing the computations in each PE and interactions with neighbor PEs. In more detail, spatial designs are a class of accelerators that can exploit high compute parallelism using communication between an array of relatively simple PEs. This type of architecture is suitable for implementing different dataflows such as reordering the computations, skipping sparsity, and various formats of data distribution among the PEs. Eyeriss [1] is a state-of-the-art energy-efficient DNN accelerator that exploits local reuse of filter weights and ifmaps to lessen memory accesses. Eyeriss [1] counts as a spatial DNN accelerator. There are also other accelerators that belong to this category [36, 37].

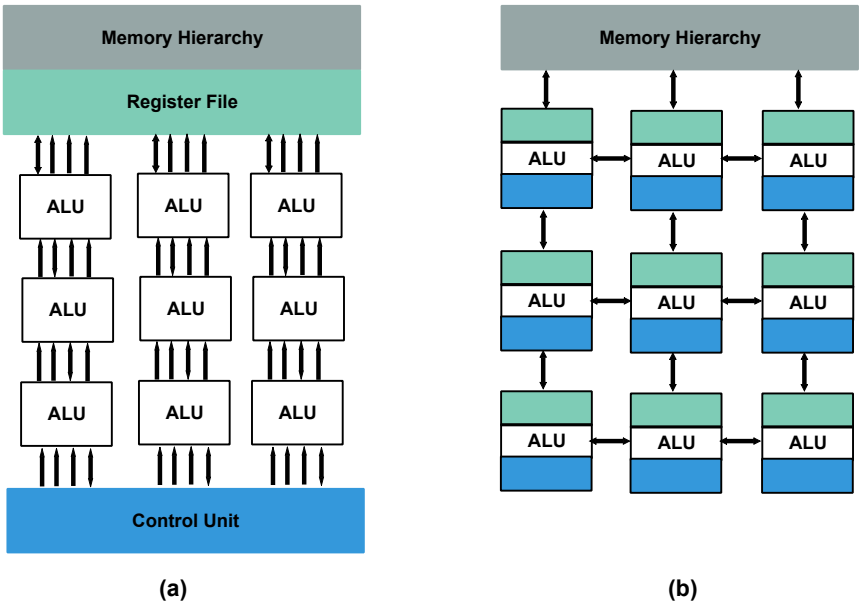


Figure 2.1: Overview of (a) temporal architecture vs (b) spatial architectures [1].

Temporal designs, also called Single Instruction Multiple Data (SIMD) architectures are another common architecture proposed for DNN accelerators. In temporal architectures, all the PEs share a control unit and a register file. One of the common temporal-based architecture is systolic array [38]. Systolic arrays are 2-D grids of homogeneous PEs that perform only nearest-neighbor communication, thus obviating the overhead for input buffering and complex routing. One of the advantages of this type of architecture is that the computation engine implementation is easier compared to spatial design. Google's Tensor Processing Unit (TPU) DNN accelerator belongs to this category [39].

## 2.2 DNN Accelerators

Regardless of the two main architectures explained in Section 2.1, there are other ways to classify contemporary DNN accelerators. For example, it is possible to classify them based on the considered techniques to improve the energy efficiency of the DNN accelerators while performing the MAC operations. Such techniques fairly fall into at least five categories depending on the property they target: dataflow, near-memory computing, pruning, data types and quantization, and approximate computing. the dataflow category itself includes a broad range of techniques such as

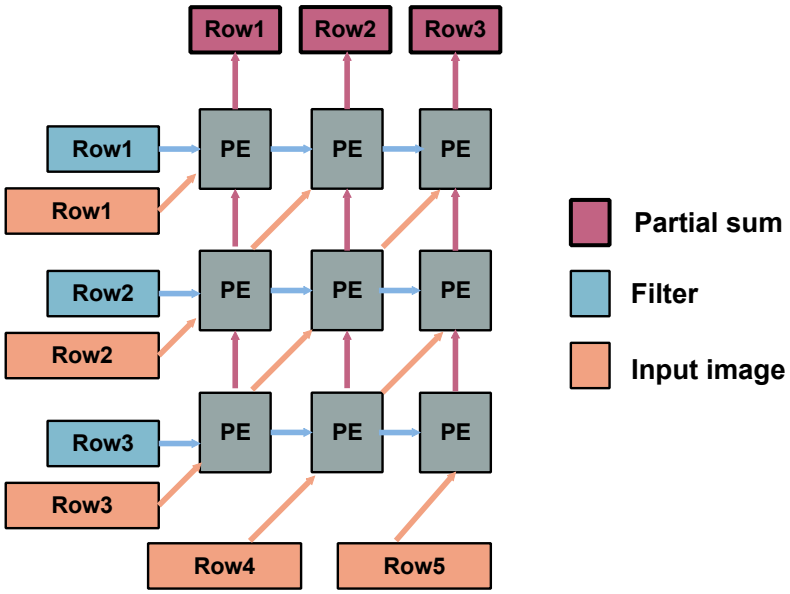


Figure 2.2: Row stationary dataflow [1]

skipping sparsity, re-configurable quantization, local data reuse. In what follows we explore some of the state-of-the-art accelerators.

Eyeriss [1] introduced row stationary dataflow which increases throughput and comes with energy efficiency. It works on both convolutional and fully-connected layers, and optimizes all types of data movement in the storage hierarchy. The overview of this dataflow is shown in Figure 2.2. As shown in Figure 2.2, input feature maps are shared diagonally, filter weights are shared horizontally and partial sums are shared vertically. In other words, all data types are reused locally. In more detail, authors in [1] showed that row stationary dataflow is more efficient compared to when only one data type is reused locally inside the computation engine. Furthermore, along with a highly flexible NoC that can adapt to a wide range of bandwidth requirements while still being able to exploit available data reuse, they can efficiently support a wide range of layer shapes to maximize the number of active PEs and keep the PEs busy for high performance. The authors in [40,41], exploit local data reuse to reduce the energy needed by data movement [40,41].

TPU [39] is an Artificial Intelligence (AI) accelerator developed by Google specifically for neural network machine learning, particularly using Google's own TensorFlow software. To add more detail, a systolic array architecture is implemented, enabling data transfer between processing

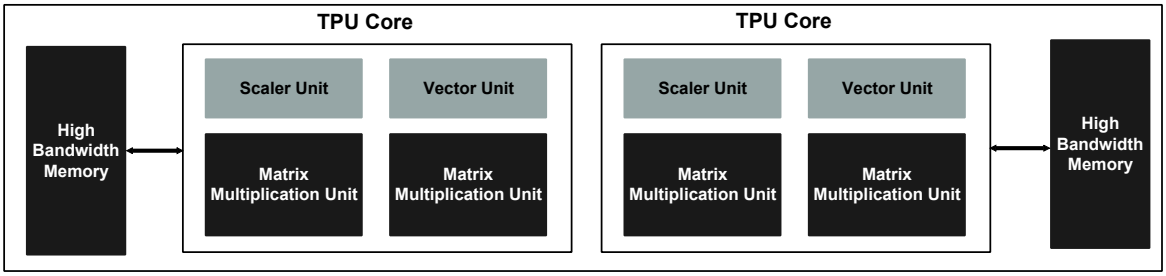


Figure 2.3: The general architecture of TPU V3 [2]

elements and thus reducing the requirement of external memory bandwidth. A single TPU [39] device contains 4 chips, each of which contains 2 TPU cores. A TPU core contains one or more Matrix Multiply Units (MXU), a Vector Processing Unit (VPU), and a Scalar Unit. The MXU is composed of 128 x 128 MACs in a systolic array. The MXUs provide the bulk of the compute power in a TPU chip. Each MXU is capable of performing 16K multiply-accumulate operations in each cycle using the bfloat16 number format. The VPU is used for general computation such as activations, softmax, and so on. The scalar unit is used for control flow, calculating memory address, and other maintenance operations. The new TPU V4 infrastructure is the fastest system ever deployed at Google. The general architecture of TPU V3 is shown in Figure 2.3.

In [3], the proposed DaDianNao architecture is a multi-chip processor architecture for machine learning which is shown in Figure 2.4. This accelerator belongs to the near memory computing category. It achieves very high internal bandwidth and low external memory communication and thus enables a high degree of parallelism during computations. The architecture of a node in this accelerator is shown in Figure 2.4 (a). A chip has 16 tiles and these tiles share 2 2MB eDRAM banks to store the inputs from the previous layer and the outputs of the current layer. The HT2.0 is used as a physical layer interface for 28nm technology. The internal architecture of a tile is shown in in Figure 2.4 (b). Each tile has 4 512kB eDRAM banks to store its weights and a Neural Functional Unit (NFU) which performs the MAC operations. The NFU needs to receive 512 B/cycle to stay busy during the classifier layer. To provide this high bandwidth while tolerating the 3 cycle eDRAM latency (at 600 MHz freq), 4 eDRAM banks are used. It is clear that a DaDianNao chip has (significant) total storage of 36MB (32MB for synaptic weights).

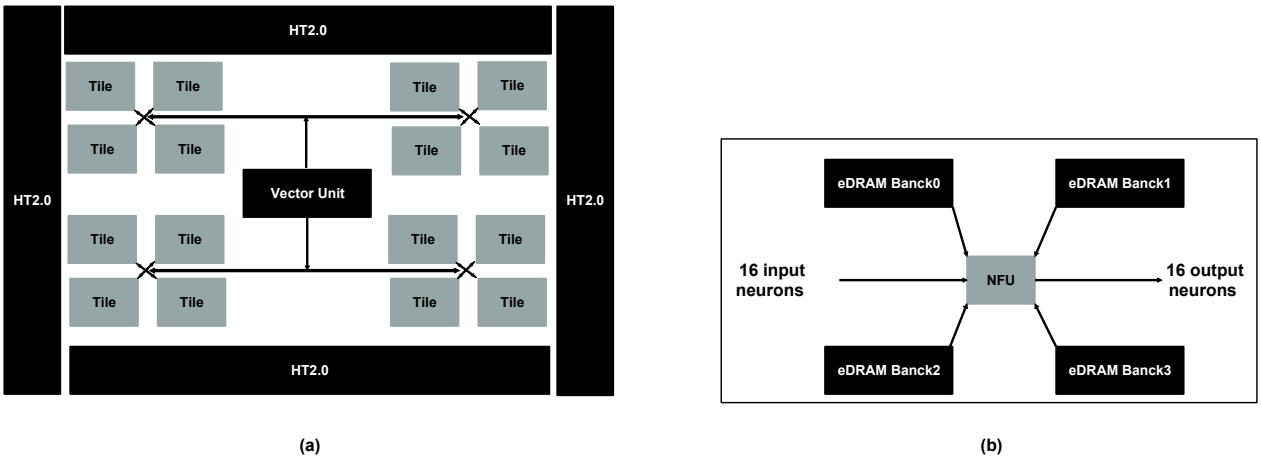


Figure 2.4: The general architecture of (a) a node and (b) a tile in [3]

As discussed earlier, modern DNNs have many parameters to provide enough model capacity, making them both computationally and memory intensive. Authors in [4] introduced an efficient compression method for pruning the large DNNs. Authors in [4] claim that their pruning method removes redundant connections and learns only important connections. Figure 2.5 shows a simple DNN after pruning. Pruning enhances inference processing time and also decreases the energy per inference required to run such large networks. The main idea in [4] is, after an initial training phase, the DNN model is pruned by removing all connections whose weight is lower than a threshold. However, the phases of pruning and retraining needed to be repeated iteratively to further compress the DNN. Authors in [4] have shown that, On the ImageNet dataset, their pruning method reduces the number of parameters of AlexNet by a factor of 9×, without incurring accuracy loss.

Furthermore, the authors in [4] introduced a weight sharing method to compress the DNNs further. They clarified their weight sharing method using an example which is shown in Figure 2.6. Assume a layer that has four input neurons and four output neurons. As shown in Figure 2.6. On the top left of this figure is a $4 \times 4$ weight matrix, and on the bottom left, there is the corresponding of gradient matrix of the weight matrix. The weights are quantized to 4 bins and all the weights in the same bin share the same value. The weights in each bin are distinguished with a unique color. In this example, using the proposed weight sharing method, instead of considering $16 \times 32$ bits for the weight matrix, it is possible to consider only $16 \times 2 + 4 \times 32$ bits. Thus for each weight, we need to store only a small index into a table of shared weights. During the backpropagation update, all

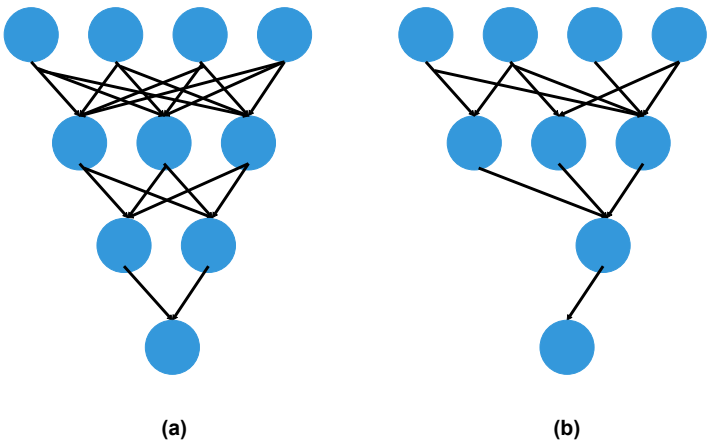


Figure 2.5: The architecture of a simple DNN (a) before pruning, and (b) after pruning [4].

the gradients are grouped by a specific color and summed together. These gradients are multiplied by the learning rate and subtracted from the shared centroids from the previous iteration. In the example shown in Figure 2.6, instead of considering $16 \times 32$ bits, there is only need to store $4 \times 32$ bits. Authors in [4] have shown that their weight sharing method can compress the weights of the convolutional layers of Alexnet to 256 shared weights which accordingly is possible to represent them using 8 bits.

As explained earlier, the complexity of DNNs is necessarily restricted by the available resources. Therefore, efforts have been made to optimize DNNs during the implementation phase in terms of resource and energy consumption. This includes both the transfer from floating-point to fixed-point arithmetic and decreasing the bit-width of the arithmetic units. The same author in [4], is evaluated the relevant energy cost of accessing different levels of memory and performing a single MAC operation both for floating-point and fixed-point with different bit-width. The result of this evaluation is reported in Figure 2.7. Based on the results shown in Figure 2.7, reducing fixed-point integer MAC units come with a significant reduction in terms of utilized energy and area. TPU [39] uses only 8-bit multipliers to perform the MAC operations. While the focus of this dissertation is on inference, it is possible to apply different quantization methods in the training phase [42–46]. In [47] the clipping range is determined by considering all of the weights in convolutional filters of a layer. In [48] the authors grouped multiple different channels inside a layer to calculate the clipping

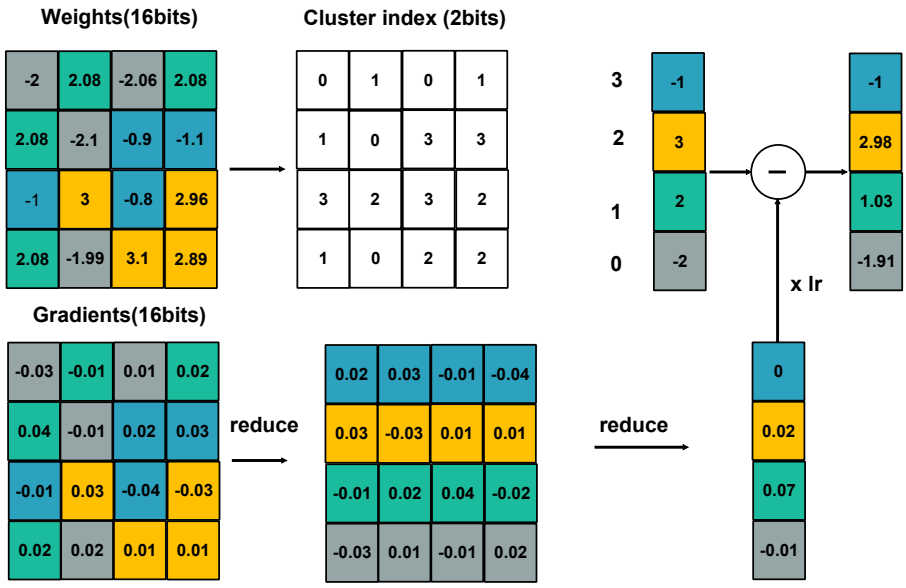


Figure 2.6: A simple example to clarify the proposed weight sharing method in [4] for a $4 \times 4$ weight matrix.

range. This could be helpful for cases where the distribution of the parameters across a single convolution/activation is possible. Authors in [5] proposed an optimization method to quantize the weights and input feature maps of their benchmark DNNs to ternary weights and 3-bit input feature maps. As an example shown in Figure 2.8, in their method the weights larger than a positive threshold are replaced with "1", all the weights smaller than a negative threshold are replaced with " $-$ 1", and the rest of the weights are replaced with "0".

Many of contemporary accelerators have targeted sparsity, that is, zero computations. Whenever, a filter weight or an input feature map is zero, that specific MAC operations can be skipped. Skipping zero computations come with saving energy and/or storage and data and control signal communications [24, 49]. The proposed architecture in [27] targets bit-wise as well as word-wise sparsity, that is zero bits since processing zero bits in a MAC operation do not affect the outcome. To add more detail, the authors convert the weights and input feature maps using Booth encoding algorithms. Then, using their novel dataflow they skip zero computation bit-wise and word-wise. The architecture which is illustrated in Figure 2.9, is a simple $2 \times 2$ format of the proposed computation engine proposed in [27]. Each PE in [27] is called Laconic PE (LPE). Since the activations

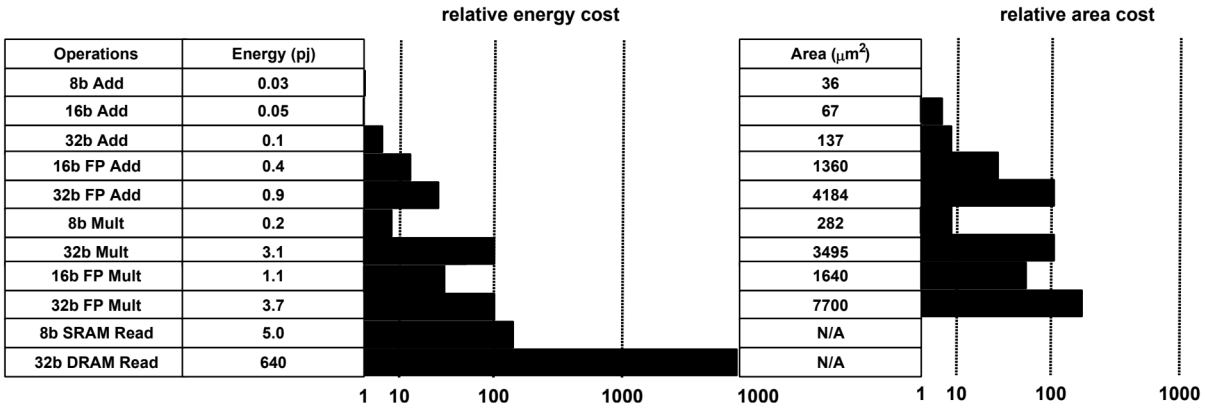


Figure 2.7: Relevant cost of accessing to different levels of memory and performing a single MAC operation in 45 nm technology.

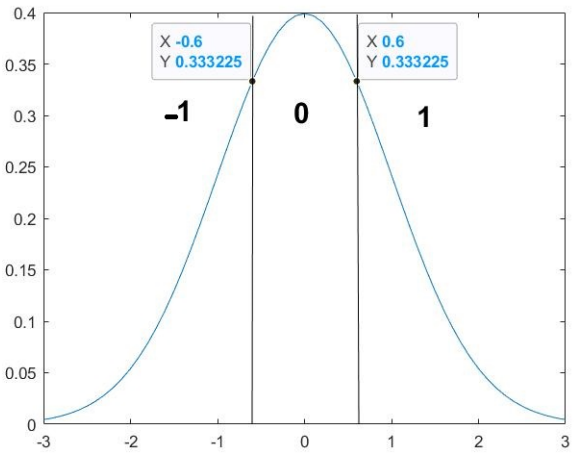


Figure 2.8: A simple example on how to convert the weights of a DNN to ternary weights [5].

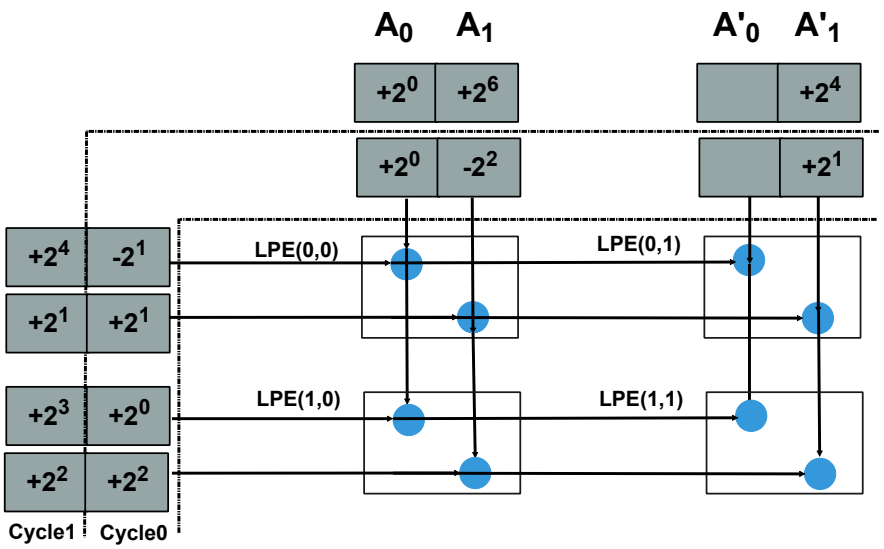


Figure 2.9: Each Laconic PE in [6] processes weights and input feature maps as a series of signed powers of two.

and filter wights are Booth encoded, each LPE processes the multiplication operations as a series of signed powers of two. In the example shown in Figure 2.9 the $W_0$ and $W_1$ are shared between the first row of LPEs and $W'_0$ and $W'_1$ are shared between the second row of the LPEs. To add more detail, regardless of skipping sparsity, they are reusing data locally in their computation engine as well. The authors in [27] showed that their architecture is flexible to perform both 8-bit and 16-bit fixed-point integer MAC operations.

In excess of regular architecture where developers consider a DRAM chip besides the processing unit, it is possible to bring compute near memory, or into memory. Accordingly, there are three different architectures which are shown in Figure 2.10. The architecture in Figure 2.10 (a) is a regular architecture which is also called digital architecture. As explained in digital format, data access is dominating. Accordingly, as explained earlier, techniques such as data reuse and network compression are proposed to decrease number of accesses to higher-level of memory. The architecture which is shown in Figure 2.10 (b) is called near memory which comes with combining DRAM banks and digital processing unit. The main advantage of such architecture is high data bandwidth. However, near memory architecture comes with expensive Analog to Digital Converters (ADCs), and Digital to Analog Converters (DACs). The architecture which is shown in Figure 2.10

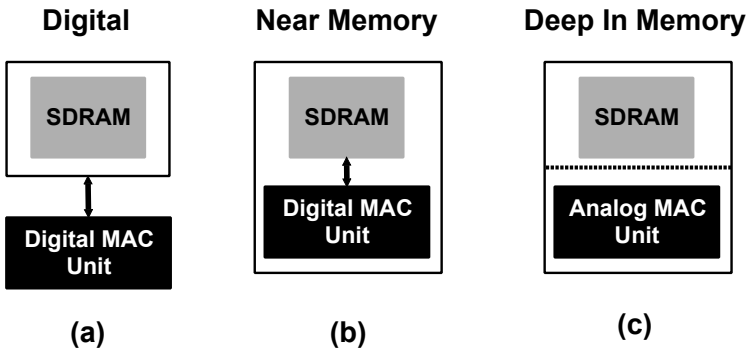


Figure 2.10: There are three general architectures for DNN accelerators based on the place of performing the computations, where in a (a) digital architecture, the processing unit and the memory chip are two different chips, in a (b) near memory architecture, the processing unit and the memory both are on the same die, in a (c) deep in memory architecture, the same technology is used for manufacturing both memory and processing unit.

(c) is called deep in memory, which includes an analog circuit to perform the computation based on voltage or current. These types of architectures also come with high bandwidth. However, these type of architectures are expensive and come with process variation problem.

The last category of optimization techniques for acceleration of DNNs that is explored in this chapter is applying approximate computing principles. Error resilience is the main factor that can determine whether approximate computing can be applied. Authors in [32] showed that one of the promising solutions to perform inference of DNNs efficiently is utilizing approximate multipliers inside the computation engine of DNN accelerators. In [34] authors have introduced approximate multipliers based on alphabet-set multiplication. In more detail, the weights are divided into parts, having 4 bits. Multiplication by each 4-bit part of the weight is implemented by shifting a pre-computed input value and followed by summation. This architecture uses an algorithm to approximate multiplication in the specified PEs of the computation engine when training DNNs. Similarly in [50], authors have proposed an algorithm that determines bit precision for all PEs to minimize power consumption for given target accuracy.

While all of the proposed algorithms in [34, 50] are very progressive and interesting research topics, that propose different paradigms of efficient DNN processing, it is not easy to integrate

them into the smart devices equipped with DNN applications. Authors in [51] have shown that the costs of applying DNNs to smart devices can be mitigated via approximate computing. In [52], the authors have evaluated how the error caused by an approximate logarithm-based multiplier affects deep DNN inferences. In more detail, the authors have explained how convolutional and fully connected layers maintain their intended functionalities despite applying a logarithm-based approximation into the computation engine of DNN accelerators. Also, It has been shown that the total computed error will not converge properly while performing the MAC operations of different layers using approximate adders. The comparison of the characteristics of the linear domain approximate multipliers and the logarithm domain multipliers are performed in [53], where approximate Booth multiplier and logarithm multiplier proposed in [53] are compared. Different versions of the approximate multipliers are compared in terms of error metrics, such as normalized mean error, and hardware metrics, such as power delay product (PDP). Like the results reported in Chapter 7 of this dissertation, in all cases, the logarithm multipliers are less accurate than the linear multiplier. In other words applying approximation in both the logarithm transform process and the logarithm domain addition process causes less accurate output.

# 3. Early Detection of Negative Output Features

Deep Neural Networks (DNNs) perform billions of computations in applications such as image classification, object detection, and image segmentation. In most well-known DNNs, an activation function follows a convolutional or a fully connected layer. Several popular activation functions involve setting all negative inputs to zero. In this chapter, we propose a novel architecture in which the activation function is merged with the prior computational layer. Our proposed dataflow can reduce the computations needed to generate a specific output feature in convolutional and fully connected layers by reordering the computation to achieve early detection of the sign of the output feature. In addition, our computation engine supports zero computation skipping, which further accelerates the layer computation. In this chapter, we build on a state-of-the-art DNN accelerator and modify it to perform early detection of negative output features. To be more specific, Section 3.2 provides a brief introduction, 3.2 introduces the contributions of our work. Section 3.3 explains the proposed computation technique. The proposed architecture is described in Section 3.4. Section 3.5 introduces our dataflow. Section 3.6 compares the performance of this work to the reference design [1] in terms of throughput and energy consumption.

---

A major portion of this chapter is published in M. Asadikouhanjani and S. -B. Ko, "A Novel Architecture for Early Detection of Negative Output Features in Deep Neural Network Accelerators," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3332-3336, Dec. 2020, doi: 10.1109/TCSII.2020.2977015.

M.A conceived of and designed the proposed accelerator. M.A carried out all the simulations, implementations, and analysis. M.A wrote the manuscript in consultation with S.K.

## 3.1 Introduction

DNNs have enhanced the accuracy of many artificial intelligence (AI) applications unprecedentedly. However, DNNs require hundreds of megabytes of storage and involve high computational complexity. There are many DNN accelerators available implemented on application specific integrated circuit (ASIC) and field-programmable gate array (FPGA) platforms [17, 18] which are better optimized in terms of power consumption compared to GPU architectures [19].

To reduce the number of MAC operations in DNNs, [20] showed that neighboring output feature map (ofmap) elements tend to have similar values and leveraged this to reduce the required MAC operations by 30.4% with a 1.7% accuracy loss. [21] equipped processing units with an approximate processing mode and achieved improvements in energy efficiency of 34.11% to 51.47% with an accuracy drop of 5%. In [22], an accelerator is proposed which performs early detection of negative output features via the use of the inverse two's complement format of the filter elements. The design presented in [22] reduces the number of required MAC operations, but the proposed architecture in [22] comes with several weaknesses:

1. No results were included in [22] for the first fully connected layer of the benchmark network since it is not possible to map this layer into the proposed accelerator.

2. The accelerator in [22] still requires extra loads and extra MAC operations in the computation engine since it performs the computations for several filters at the same time for all the layers. Additionally, the dataflow in [22] does not support skipping sparsity.

3. The dataflow proposed in [22] only works when the ifmaps are positive or zero. Therefore, there is no speedup over the first convolutional layer of the network.

Near memory computation aims to minimize expensive data movement, a bottleneck in data-centric applications [54]. Disadvantages of such techniques include the use of analog-to-digital converters or digital-to-analog converters which occupy a large chip area, as well as non-generic layouts which are expensive to build.

Eyeriss [1] is a state-of-the-art energy-efficient DNN accelerator that exploits local reuse of filter weights and ifmaps to lessen memory accesses. Although the dataflow proposed in [1]

is energy-efficient, it still performs unnecessary loads and computations for the negative output features, meaning that there is an opportunity to optimize the dataflow and improve performance. The interal architecture of Eyeriss is used as a basis for this work.

## 3.2 Contributions of this chapter

1. A novel architecture for the early detection of negative output features in hardware is proposed. The introduced method reduces the average number of MAC operations by 10.64% and the average number of loads by 3.86% while maintaining accuracy.

2. The proposed dataflow can map various fully connected and convolutional layers and keeps the computation engine busy with effectual computations, achieving an average speedup of $\times$2.19 and an average improvement in energy efficiency of $\times$1.94.

3. It is possible to integrate our proposed computation technique into any real-time DNN accelerator (both training and inference platforms) to achieve further optimization.

## 3.3 Proposed Computation Technique

Many modern DNNs such as AlexNet [55] and VGG-16 [15] apply the rectified linear unit (ReLU) activation function to the outputs of both fully-connected and convolutional layers, which thresholds all negative values in the ofmap to zero. Since negative ofmap values are not used, performance can be improved if operations are reordered to skip the computation of negative output features. The percentage of negative output features in different convolutional layers of AlexNet is evaluated using the ImageNet validation dataset. The results of this evaluation are shown in Figure 3.1, which illustrates the opportunity for improving performance by reordering computation.

This work proposes a novel method of reordering the computations in a spatial computation engine which is illustrated in the example shown in Figure 3.2. The dataflow involves two rounds of computation: all input features corresponding to positive weights are loaded for all channels in the first round, while the input features corresponding to negative weights of each channel are loaded gradually in the second round. In the example shown in Figure 3.2, there is no need to continue the MAC operations for channel 1 of the ifmaps in the second round of computation. Therefore, the

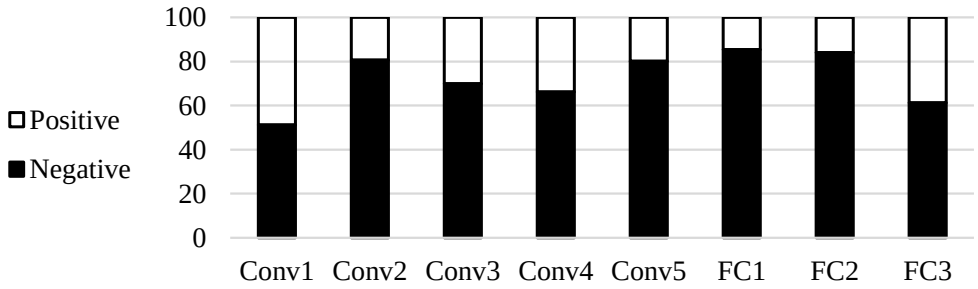


Figure 3.1: Sign-based distribution of output features of various layers of AlexNet.

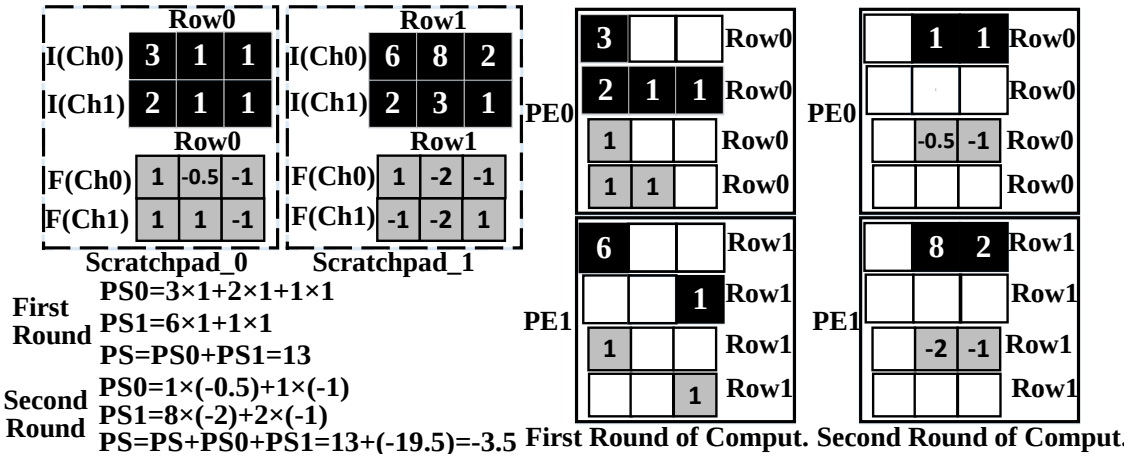


Figure 3.2: A 2-D reordered convolution example in a spatial computation engine with two PEs.

number of load and MAC operations is reduced by 3 compared to regular dataflow.

According to our proposed dataflow and the configuration of the spatial units, the main control unit loads all channels of filters and ifmaps used for computing a few specific output features into different groups of processing elements (PEs). The control unit inside each PE uses two tables for reordering the computations and skipping zeros. The first table is loaded with the sign bits of each loaded filter element and the second table is loaded with the flag bits indicating zero-valued ifmap elements.

The main control unit also loads a larger 3.1 kB table which is contained within the global buffer, with the sign bits of the loaded filter elements in the computation engine. Using this table, the main control unit can reorder the computations for all the layers of benchmark networks. When the size of ifmap rows for a layer is larger than the size of the ifmap scratchpads in the computation engine, the main control unit starts with loading and doing the computations for the filter rows

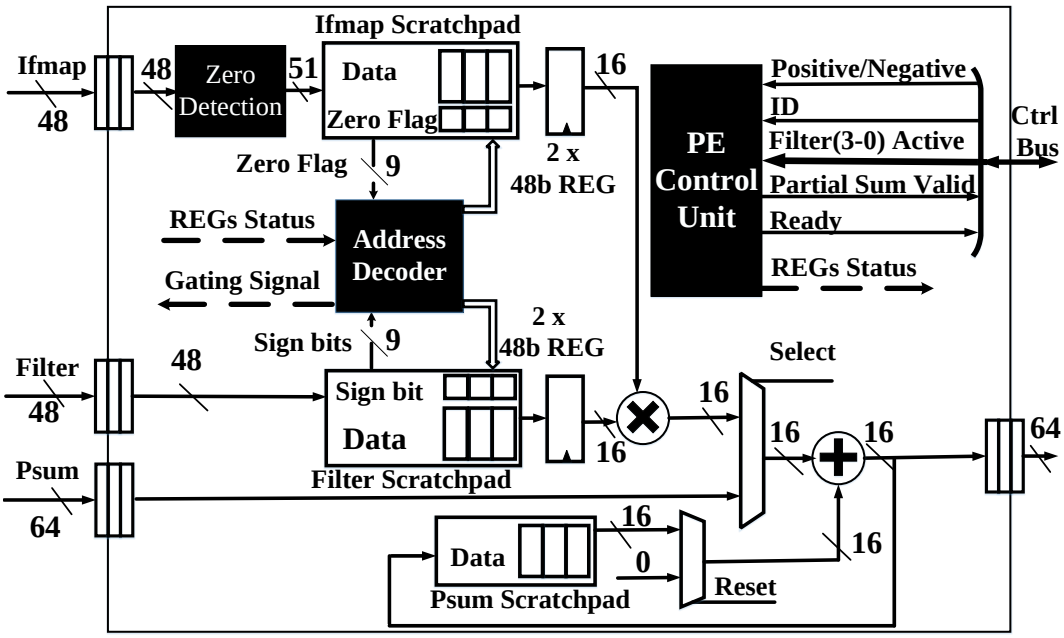


Figure 3.3: The internal architecture of the proposed PE.

that contain more than one positive element and their corresponding ifmap rows. Then, the main control unit continues the computations for the rest of the filter rows and their corresponding ifmap rows.

## 3.4 Proposed Architecture

### 3.4.1 PE Specification

The 16-bit fixed-point architecture of the proposed PE is shown in Figure 3.3. Each PE has a 24×16b partial sum (psum) scratchpad, three 27×16b ifmap scratchpads, a 9×9b zero scratchpad, three 78×15b filter scratchpads, and a 26×9b sign scratchpad. An address decoder unit fetches ifmaps according to the zero scratchpad and likewise fetches filters according to the sign scratchpad. Two 48b registers are used to store these newly-fetched rows of ifmaps and filters. A 16×16b multiplier is used for computation. Each PE has a control unit that coordinates the computation process within that PE.

### 3.4.2 Loading Ifmaps and Filters Into the PEs

When loading new ifmap rows into the three ifmap scratchpads, the zero detection unit sticks one flag bit to each ifmap element indicating whether that ifmap element is zero. Each row of the zero scratchpad contains the flag bits for the elements of three different rows of different channels stored in the ifmap scratchpads. When loading the filter rows into the three filter scratchpads, the PE control unit uses the sign scratchpad to store the sign bits of the filter elements. Each row of the sign scratchpad contains the sign bits of the elements of three different rows of different channels of loaded filters.

### 3.4.3 Address Decoder Unit

The address decoder unit is contained within the control unit of each PE. The address decoder unit uses one row of the zero scratchpad and one row of the sign scratchpad to skip zero computations and reorder the computations in each cycle. For example, during the positive subset of computations, the address decoder unit skips negative filter weights in each row and simultaneously checks for any non-zero elements available in the corresponding row of the ifmap scratchpad. After completing this round of computation, the PE begins the computations for negative filter weights, which are split across several processing cycles. During the second round of computation, the main control unit considers the sign of total psum at the end of each processing cycle. If the total psum computed so far is negative, it is evident that the output feature will be negative and the main control unit will prevent further computation.

The address decoder unit additionally uses the two 48b registers as a small ping-pong buffer. While the unit is performing the computations for the loaded row of ifmap and filter inside the first set of 48b registers, the next row of ifmaps and filters is loaded into the second set of registers.

### 3.4.4 MAC Operation

The multiplier performs the computation for every 16 bits of the 48b register provided by the address decoder unit as indicated by the Partial Sum Valid signal. The control unit of each PE stores 16 bits of the sum of products for each filter into the psum scratchpad.

Table 3.1: Energy cost of basic operations for the proposed accelerator.

| Operation | Energy (pJ) | Normalized Energy Cost |
|---|---|---|
| 16b Fixed Point Add | 0.0865 | 1 |
| 16b RF Access | 0.3832 | 4.43 |
| 16b Fixed Point Mult | 2.0783 | 24.02 |
| 16b RF Inter PE Access | 0.75 | 8.67 |

Table 3.2: Mapping parameters for proposed dataflow.

| Parameter | Description |
|---|---|
| $r$ | Number of PE sets that process different channels |
| $t$ | Number of different filters that each PE process |
| $q$ | Number of channels processed by a PE set |

### 3.4.5 Computation Reordering by the Main Control Unit

The main control unit maps the computation engine into different groups of PEs for different layers of the benchmark network. This unit uses the Ready signal of each PE for loading all the PEs in a group. The main control unit distinguishes the first and the second subset of computation with the Positive/Negative signal of each PE. The computation inside a group of PEs is complete when all the PEs of that specific group are done their computations in a processing cycle. If the sign of the total psum for a specific part of the ifmap and a specific filter is negative, the main control unit avoids extra computation for that part of the ifmap by using the Filter (3-0) Active signal. The main control unit stores a bit for each output feature which indicates whether the computation engine should continue to process the computations for that specific output feature. The PE skips the computations for a specific output feature using the valid addresses of the filter scratchpad.

### 3.4.6 Computation Engine Size and NoC Specifications

A 12×15 computation engine and a 108 kB global buffer are considered for both our design and Eyeriss. The Network-on-Chip (NoC) which manages the data delivery between the global buffer

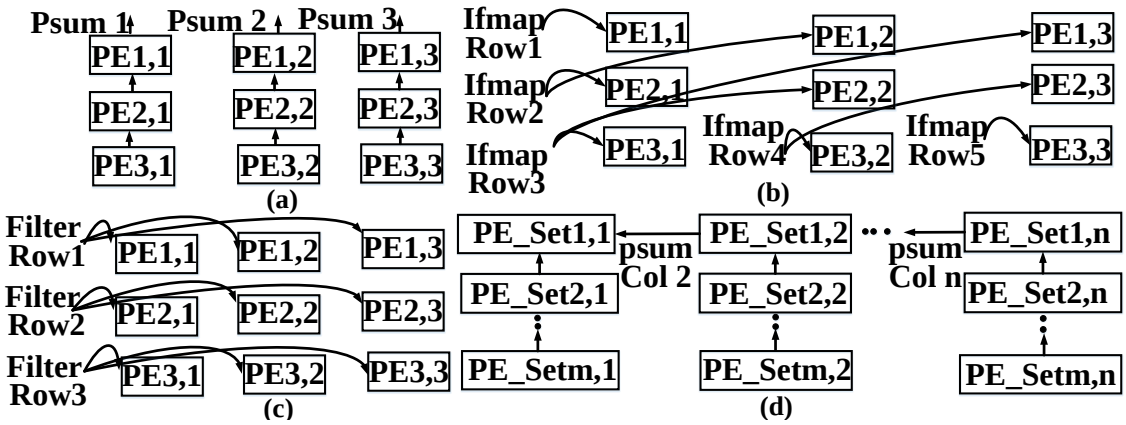


Figure 3.4: Local reuse of data via (a) vertical reuse of psums, (b) diagonal reuse of ifmap rows, (c) horizontal reuse of filter rows, and (d) horizontal reuse of psums across the first row of the computation engine.

and the computation engine is similar to that of [1] but with some key differences:

1. The width of the ifmap data bus and the filter data bus is 48b which makes the implementation of our dataflow more straightforward.

2. The PEs in the first row of the computation engine share their partial sums through local horizontal connections.

3. The control signals introduced in the previous section are added to the control signal bus.

## 3.5 Proposed Dataflow

### 3.5.1 Energy Cost of Basic Operations and Data Movement

The energy cost of basic functions and data movements in our accelerator for a frequency of 1Ghz are shown in Table 3.1. According to [1], the normalized energy cost of a global buffer access (>100kB) implemented in 65nm technology is 6 times larger than that of a MAC operation. Table 3.1 illustrates the importance of decreasing the number of computations, reducing the number of memory accesses, and maximizing local data reuse.

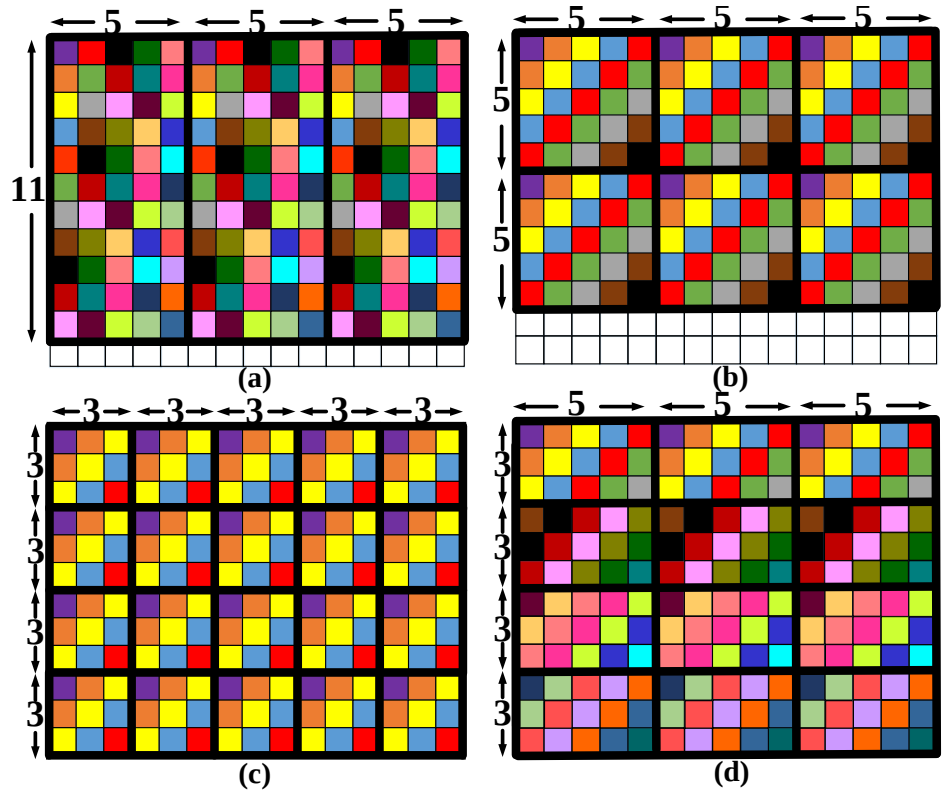


Figure 3.5: Different configurations of the computation engine for calculating (a) 5 output features for $11 \times 11$ filters and stride 4, (b) 5 output features for $5 \times 5$ filters and stride 1, (c) 3 output features for $3 \times 3$ filters and stride 1, and (d) 3 output features for $3 \times 3$ filters and stride 1.

### 3.5.2 2-D PE Sets and Processing Pass

A 2-D PE set is composed of several rows of PEs, and the computation engine can be configured to fit several PE sets. The parameters used to describe the configuration of the computation engine are introduced in Table 3.2. The main control unit performs the computations for different channels of $t$ different filters using $r$ number of PE sets simultaneously. Every $r$ sets accumulate their partial sums for each filter together. Additionally, $q$ specifies the number of ifmap channels that each PE processes. Processing Pass refers to the configuration of PE sets inside a computation engine to calculate multiple 2-D convolutions of different channels of different filters simultaneously.

### 3.5.3 Local Reuse of Ifmap Rows, Filter Rows, and Psums

The reuse of ifmap rows, filter rows, and psums in the proposed accelerator are shown in Figure 3.4 (a-c). As shown in Figure 3.4 (d), the PE sets in the first row of the spatial array also reuse the partial sums horizontally.

### 3.5.4 Computation Engine Configuration and Mapping Formats

The dataflow format of [1] would require loading the ifmap twice, once for each computation round, increasing the number of global buffer accesses. This provides the motivation for the use of the proposed dataflow in our accelerator. Four configuration formats are proposed for use with our dataflow and are illustrated in Figure 3.5. Bold black lines distinguish each 2-D PE set. Inside each 2-D PE set, the PEs with the same ifmap channels share the same color. The PE sets with the same set of colors perform the convolution computation of a specific part of ifmap channels and the channels of a few filters. For example, the formats in Figure 3.5 (a-c) show the PE sets that perform the convolution for a specific part of ifmap channels, but the format in Figure 3.5 (d) shows the PE set that performs the convolution for different parts of ifmap channels.

### 3.5.5 Convolution Direction and Global Buffer Access

The main control unit loads different parts of the ifmap channels in a depthwise direction and loads the filter channels sequentially. The main control unit continues the computation by loading different parts of the ifmaps vertically.

Reusing psums horizontally and doing the convolution in a depthwise direction decreases the number of global buffer accesses needed for the psums. The number of global buffer accesses for ifmaps and filters are the same as [1].

### 3.5.6 PE Set Synchronization

During the second subset of computations, the main control unit specifies the number of channels for which each PE must perform the computation. Then, using the Partial Sum Valid signal of each PE, this unit synchronizes all the PE sets for partial sum calculation and sign monitoring.

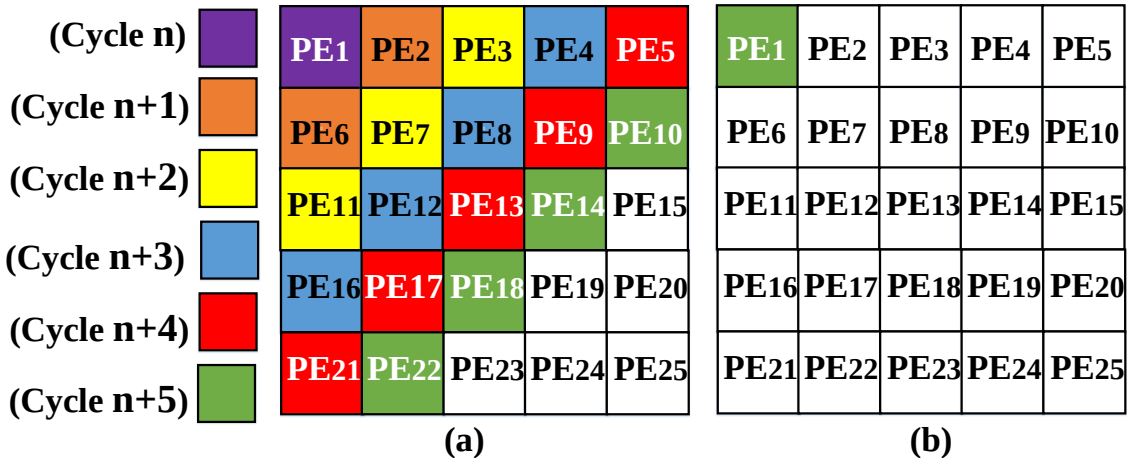


Figure 3.6: Ifmap reuse while loading the PE sets via (a) reusing the ifmaps diagonally, and (b) reusing the ifmaps in a depthwise method.

## 3.6 Results and Discussion

### 3.6.1 Mapping Benchmark Networks to the Proposed Accelerator

Two popular image classifiers are used for evaluating our accelerator, namely AlexNet and VGG-16. Conv1 layer of AlexNet is mapped according to the Figure 3.5 (a) format. Conv2 layer is mapped according to the Figure 3.5 (b) format. Conv3-Conv5 layers are mapped according to the Figure 3.5 (c) format. All the convolutional layers of VGG-16 are mapped according to the Figure 3.5 (c) format, except Conv1_1 which is instead mapped according to the Figure 3.5 (d) format. All fully connected layers are mapped according to the Figure 3.5 (c) format but without any local reuse of ifmaps. The percentage of active PEs in the Figure 3.5 (a-d) configuration formats is 91.6%, 83.3%, 100%, and 100%, respectively. In our dataflow, the number of Processing Passes inside each PE for different configurations equals the number of loaded filters.

While performing the computations for the first convolutional layer for both benchmark networks, the control unit inside each PE uses the zero scratchpads for tracking the sign bits of the ifmap elements because the ifmap elements of these layers could be negative. For the first fully connected layer of both benchmark networks, the main control unit starts with loading and doing the computations for the filter rows that contain more than one positive element and their corresponding ifmap rows. Then, this unit continues the computations for the rest of the filter rows and their

corresponding ifmap rows. The ifmaps and filters for all fully connected layers are transformed into rows containing three elements.

Figure 3.6 shows the reusing of ifmaps for the PE set with configurations according to the Figure 3.5 (b) format. In each PE, the considered ifmap scratchpads have two parts. As Figure 3.6 (a) shows, the main control unit at cycle $n$, loads the data into the first part of the ifmap scratchpads for PE1. Figure 3.6 (b) shows how this unit, at cycle $n + 5$, loads the second part of the ifmap scratchpads of PE1 and the first part of the ifmap scratchpads in PE10, PE14, PE18 and PE22 with the same ifmap channels.

### 3.6.2 Implementation

A cycle-accurate simulator is created to model both the Eyeriss architecture and our proposed design. Both designs are synthesized using Synopsys Design Compiler and TSMC 65nm technology library. ARM Artisan 65nm SRAM and register file generator are used for creating the global buffer and scratchpads. The clock frequency is 200MHz and the core voltage is 1.1V for both designs.

### 3.6.3 Hardware Performance Evaluation

The percentage reduction in MAC operations and the achieved speedup of this work over [1] are shown in Tables 3.3 and 3.4. For a more fair comparison, The speedup is reported in two modes since the dataflow of Eyeriss does not support skipping sparsity. The proposed dataflow does not support skipping sparsity in mode 1, but skips over zero computations in mode 2. The average speedup of the first and second modes are $\times$1.92 and $\times$2.19 respectively. The average reduction in number of loads is 3.59% for AlexNet and 4.13% for VGG-16.

The throughput and energy consumption comparison of this work to Eyeriss [1] are shown in Table 3.5. The area overhead of applying our method to each PE is 11.9% and in total is 10.9%. The average power consumption of [1] and our proposed architecture is 292.4 mW and 327.2 mW respectively. The computation engines account for 87% and 88.1% of the consumed power in [1] and our design respectively. The average energy consumption of [1] and our design for one inference while running AlexNet is 28.7 mJ and 13.8 mJ and while running VGG-16 are 418.4 mJ and 230 mJ, respectively.

Table 3.3: Reduction in MAC operations and speedup of the proposed dataflow compared to [1] while running AlexNet.

| Layer | $r$ | $t$ | $q$ | MAC Op. Reduction | Speedup | |
|---|---|---|---|---|---|---|
| | | | | | Mode 1 | Mode 2 |
| Conv1 | 3 | 2 | 1 | 21.77% | 1.78 | 1.78 |
| Conv2 | 6 | 1 | 8 | 8.65% | 1.75 | 2.01 |
| Conv3 | 20 | 4 | 13 | 6.54% | 1.82 | 2.15 |
| Conv4 | 20 | 4 | 10 | 9.90% | 2.06 | 2.36 |
| Conv5 | 20 | 4 | 10 | 18.26% | 2.18 | 2.38 |
| FC1 | 20 | 1 | 13 | 8.78% | 1.69 | 2.02 |
| FC2 | 20 | 1 | 13 | 13.15% | 1.61 | 2.06 |
| FC3 | 20 | 1 | 13 | 7.36% | 1.3 | 1.63 |

To provide a fair evaluation of our proposed accelerator, the accelerator in [22] is simulated and explored. Since the computation engine and the implementation technology used in [22] are different than that used in our implementation, the throughput of both accelerators are compared based on the number of MAC operations needed for one VGG-16 inference. The results show that the inference per MAC operation of our work is 1.7 times higher than [22]. The average number of load operations for one inference in our work is 4.17% less than [22].

Table 3.4: Reduction in MAC operations and speedup of the proposed dataflow compared to [1] while running VGG-16.

| Layer | $r$ | $t$ | $q$ | MAC Op. Reduction | Speedup | |
|---|---|---|---|---|---|---|
| | | | | | Mode 1 | Mode 2 |
| Conv1_1 | 12 | 4 | 1 | 11.03% | 1.2 | 1.2 |
| Conv1_2 | 20 | 4 | 4 | 7.49% | 1.16 | 1.18 |
| Conv2_1 | 20 | 4 | 4 | 6.41% | 1.07 | 1.09 |
| Conv2_2 | 20 | 4 | 7 | 6.8% | 1.43 | 1.5 |
| Conv3_1 | 20 | 4 | 7 | 4.95% | 1.44 | 1.53 |
| Conv3_2 | 20 | 4 | 13 | 6.88% | 2 | 2.11 |
| Conv3_3 | 20 | 4 | 13 | 9.88% | 2.15 | 2.2 |
| Conv4_1 | 20 | 4 | 13 | 8.71% | 2.02 | 2.15 |
| Conv4_2 | 20 | 3 | 13 | 8.9% | 3.06 | 3.13 |
| Conv4_3 | 20 | 3 | 13 | 14.11% | 2.97 | 3.34 |
| Conv5_1 | 20 | 3 | 13 | 10.1% | 3.14 | 3.71 |
| Conv5_2 | 20 | 3 | 13 | 12.01% | 3.22 | 3.78 |
| Conv5_3 | 20 | 3 | 13 | 20.67% | 3.3 | 3.9 |
| FC1 | 20 | 1 | 13 | 10.8% | 1.8 | 2.1 |
| FC2 | 20 | 1 | 13 | 9.4% | 1.88 | 2.71 |
| FC3 | 20 | 1 | 13 | 4.6% | 1.44 | 1.85 |

Table 3.5: Comparison of throughput and energy consumption

| Hardware Specs. | Eyeriss [1] | | This work | |
|---|---|---|---|---|
| Area (logic only) | 1314K (NAND2) | | 1560K (NAND2) | |
| Scratchpads | 119kB | | 121kB | |
| SRAM | 108kB | | 108kB | |
| PEs | 180 | | 180 | |
| Performance Metric | AlexNet | VGG-16 | AlexNet | VGG-16 |
| Batch Size | 1 | 1 | 1 | 1 |
| Inference/S | 10.14 | 0.7 | 23.73 | 1.42 |
| Inference/J | 34.74 | 2.39 | 72.35 | 4.35 |

# 4. Enhancing the Utilization of PEs in Spatial DNN Accelerators

Equipping mobile platforms with deep learning applications is very valuable. Providing health-care services in remote areas, improving privacy, and lowering needed communication bandwidth are the advantages of such platforms. Designing an efficient computation engine enhances the performance of these platforms while running Deep Neural Networks (DNNs). Energy-efficient DNN accelerators use skipping sparsity and early negative output feature detection to prune the computations. Spatial DNN accelerators in principle can support computation-pruning techniques compared to other common architectures such as systolic arrays. These accelerators need a separate data distribution fabric like buses or trees with support for high bandwidth to run the mentioned techniques efficiently and avoid Network on Chip (NoC) based stalls. Spatial designs suffer from divergence and unequal work distribution. Therefore, applying computation-pruning techniques into a spatial design, which is even equipped with a NoC that supports high bandwidth for the Processing Elements (PEs), still causes stalls inside the computation engine.

In a spatial architecture, the PEs that perform their tasks earlier have a slack time compared to others. In this cahpter, an architecture with a negligible area overhead based on sharing the scratchpads in a novel way between the PEs is proposed to use the available slack time caused by applying computation-pruning techniques or the used NoC format. With the use of our dataflow, a spatial engine can benefit from computation-pruning and data reuse techniques more efficiently.

---

A major portion of this chapter is published in M. Asadikouhanjani and S. -B. Ko, "Enhancing the Utilization of Processing Elements in Spatial Deep Neural Network Accelerators," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1947-1951, Sept. 2021, doi: 10.1109/TCAD.2020.3031240.

M.A conceived of and designed the proposed accelerator. M.A carried out all the simulations, implementations, and analysis. M.A wrote the manuscript in consultation with S.K.

To be more specific, Section 4.2 explains the proposed computation techniques. The proposed architecture is described in Section 4.3. Section 4.4 introduces our dataflow. Section 4.5 compares the performance of this work to the reference design in terms of throughput and energy consumption.

## 4.1 Introduction

While DNNs deliver state of the art accuracy on Artificial Intelligence (AI) tasks, they come with hundreds of megabytes of storage and billions of computations. Different architectures have been proposed for accelerating DNNs on various platforms until now [17, 56, 57]. Pruning the needed Multiply and Accumulate (MAC) operations of DNNs is one of the main features of energy efficient DNN accelerators [58].

ReLU is still the most popular activation function for many DNN models. This function adds non linearity to the networks by replacing the negative outputs of computation-heavy layers with zero. It is possible to cut the computations short if the main control unit determines that the output will be negative. The same thing is true for tanh activation function by replacing the outputs lower than a considered negative threshold with -1. In [58], the authors of this cahpter proposed a real-time architecture for early detection of negative output features in spatial accelerators. The computations are reordered using sign tables inside and outside of the computation engine. This method decreases the number of MAC operations and the number of memory accesses. This accelerator is built on Eyeriss [1]. The used NoC format in [58], like Eyeriss cannot provide the needed bandwidth of the PEs which causes stalls in the computation engine.

Authors in [23] introduced NoC-based DNN accelerator. They showed that efficient PE and NoC designs are both critically important to keep the system performance high in the NoC-based DNN accelerators. Compared with the conventional DNN accelerators, the number of memory accesses in NoC-based design is reduced by 88% up to 99%.

Eyeriss V2 [24] is a state-of-the-art energy-efficient DNN accelerator that exploits a flexible NoC which provides the required bandwidth of the computation engine while running various DNNs. With the use of the proposed NoC types in [23] and [24] and data reuse techniques, it is possible to decrease the NoC based stalls in sparse DNNs. However, these accelerators perform

non-effectual computations for negative outputs. Furthermore, even by providing such NoC for the PEs, applying computation-pruning techniques still causes stalls inside spatial designs.

In this cahpter, An architecture is proposed that overcomes the mentioned problems and increases the utilization of PEs inside the spatial design. The contributions of this chapter are:

1. A novel real-time architecture for the spatial DNN accelerators is proposed which uses the slack time caused by computation-pruning techniques and the used NoC format that achieves a speedup of $\times 1.24$ compared to reference design, in terms of performing one inference.

2. An energy efficiency per inference of $\times 1.18$ is achieved with only 13.53% and 1.68% area overhead compared to the consumed area of the logic and the whole reference design respectively. This improvement comes without changing the number of PEs and the considered bandwidth.

3. Using the proposed sharing technique, all the PEs remain active while performing the computations of convolutional layers with different filter sizes and depth-wise and $1\times 1$ convolutional layers. This issue leads to performance enhancement of the computation engine.

## 4.2 Proposed Computation Technique

To evaluate the mentioned problems above, an architecture which is widely used by spatial designs is chosen and explored [3, 59]. Figure 4.1 shows a $2\times 2$ version of the chosen computation engine which is equipped with horizontal and vertical 1D multi-cast NoC types. Less needed bandwidth and high data reuse are the main features of this type of NoC compared to other types such as uni-cast and 1D systolic. In this architecture, the PEs are loaded by stored Input feature map (Ifmap) and filter rows inside the Ifmap and filter Global Buffers (GBs).

Figure 4.2 shows how the mentioned stalls occur in a $2\times 2$ version of the chosen spatial architecture by an example. In this example, each PE cluster has only one PE. The computations are performed for two channels of Ifmaps and filters. The Ifmap channels are loaded in an order that is determined by the number of negative weights in their corresponding filter channels. The Ifmap channels that their corresponding filter channels have the least number of negative weights

are loaded first. In addition, one clock cycle is considered for each MAC operation in each cluster. Figure 4.2 (a) shows the computations for the first channel and Figure 4.2 (b) shows the computations for the second channel of two different Ifmap parts and two different filters. In this example, the clusters in the second column are idle during cycles 0-3 because the values of the first channel of Ifmap_1 are zero. In addition, the clusters in the first row are idle during cycles 4-7 because their Partial Sum (PSum) value is negative or zero by the third cycle and the elements of the second channel of their corresponding filter are negative. Therefore, the PSum values of the clusters in the first row will be negative. In this spatial design, the total computation delay is eight which equals to computation delay of a cluster with highest number of MAC operations.

Figure 4.3 shows the dataflow in different clusters of the given example. Figure 4.3 (a) shows the computation delay of the clusters when the scratchpads are not shared. Figure 4.3 (b) shows the computation delay of the clusters when the scratchpads are shared diagonally. Each cluster and its associated computations are distinguished with different colors in Figure 4.3. Without sharing the scratchpads, the total computation delay by using early negative output detection technique is eight. As shown in Figure 4.3, clusters 1 and 3 have four clock cycles as slack time while performing the computations of the first Ifmap and filter channel. These clusters can cooperate with clusters 2 and 0 during the available slack time. By sharing scratchpads diagonally, clusters 1 and 3 perform the computations of row 1 of the first channel of cluster 2 and 0 respectively. Therefore, the total computation delay of the first channel decreases from four to two. In addition, clusters 0 and 1 can cooperate with clusters 3 and 2 as well while performing the computations of the second channel. Accordingly, by sharing the scratchpads, the total computation delay of both channels decreases from eight to four. Sharing the scratchpads diagonally which is very efficient in terms of hardware implementation provides the chance of using the slack time between different columns and rows of the computation engine at the same time.

The occurred stalls in the above example are caused by applying computation-pruning techniques. For NoC based delay stalls, the same thing is true. In fact, the PEs that receive their data sooner will finish their task earlier. These PEs can cooperate with the other PEs to increase the performance of the whole spatial computation engine. In what follows, the architecture of the proposed Twin-PE is explained.

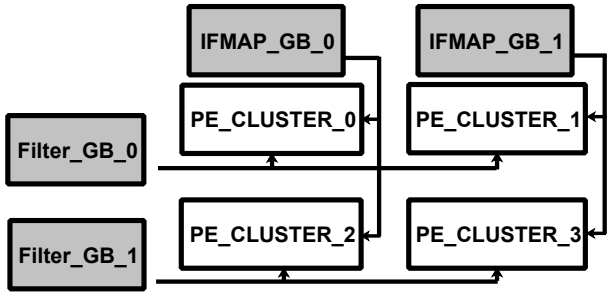


Figure 4.1: A 2×2 computation engine with horizontal and vertical 1D multi-cast NoC.

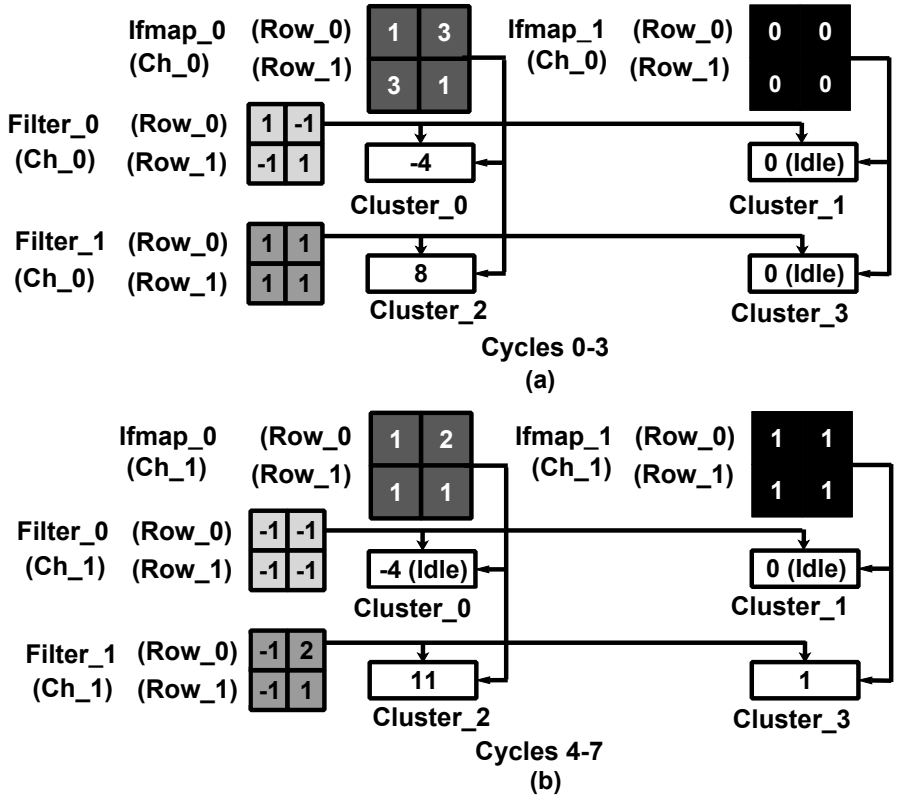


Figure 4.2: Idleness of clusters in a 2×2 computation engine, where (a) shows the computations in cycles 0-3, and (b) shows the computations in cycles 4-7.

## 4.3 Proposed Architecture

### 4.3.1 MAC Unit Specifications

Each single PE in our design is a modified version of the PE proposed in [58]. To share the computations, the logic that performs Multiply and Accumulate (MAC) operation and the Ifmap

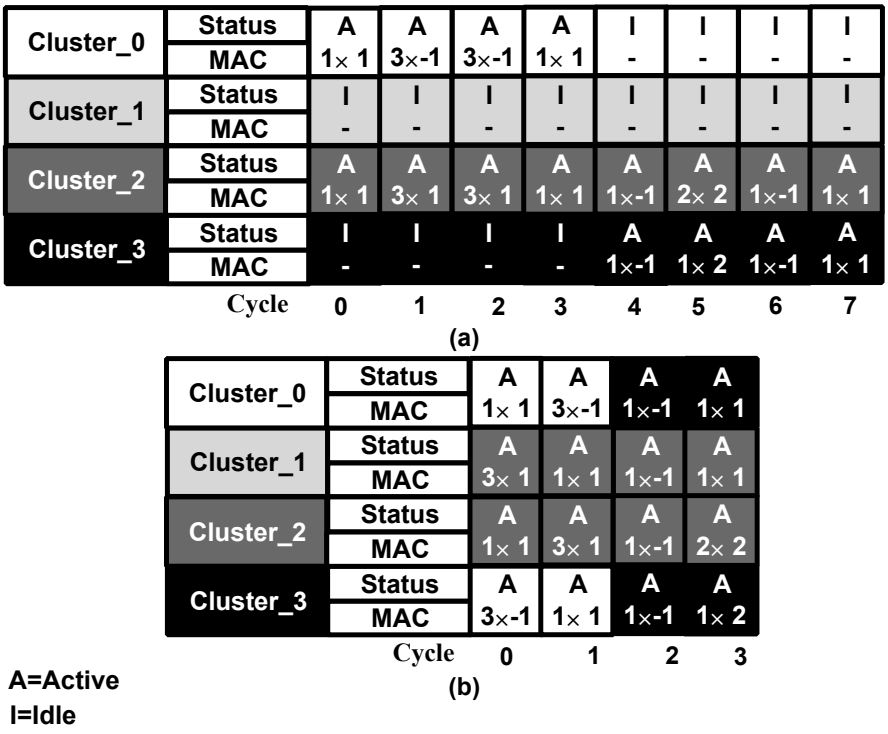


Figure 4.3: The status of clusters and their dataflow when (a) the scratchpads are not shared compared to when (b) the scratchpads are shared.

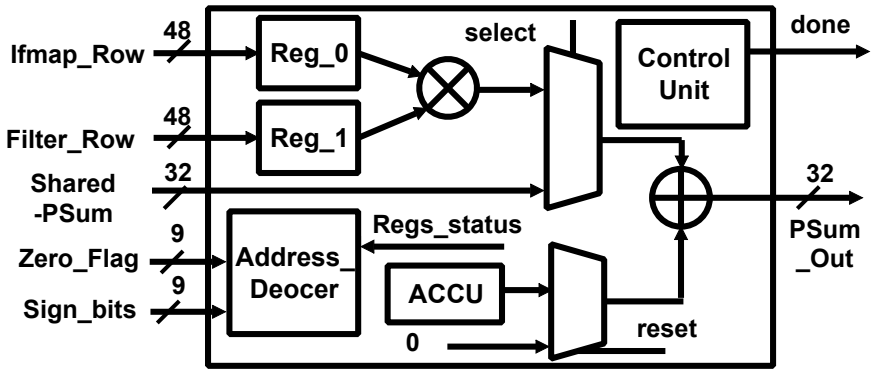


Figure 4.4: The internal architecture of the proposed MAC unit.

and filter scratchpads are separated. The 32-bit fixed-point MAC unit is shown in Figure 4.4. In each MAC unit, an address decoder unit fetches the Ifmap rows according to the zero scratchpad and likewise fetches filters according to the sign scratchpad. Two 48b registers are used to store these newly-fetched rows of Ifmaps and filters. Shared PSum port in each MAC unit is a PSum created by the other MAC unit in a Twin-PE. A 16×16b multiplier is used for computation. Each

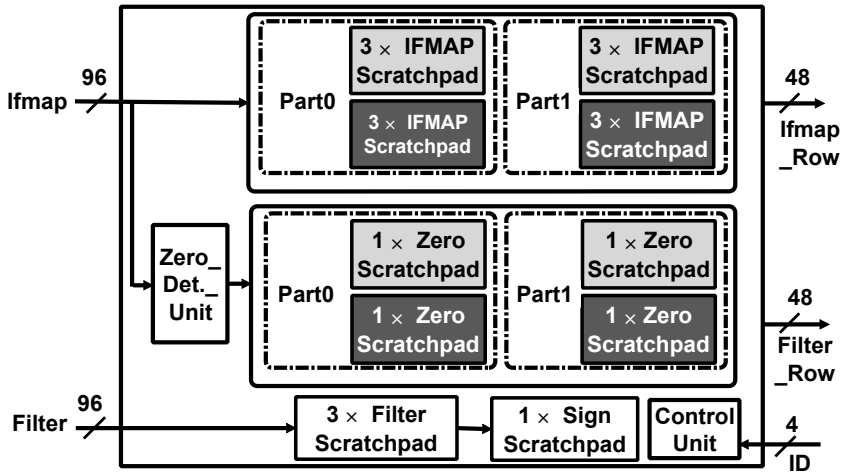


Figure 4.5: The internal architecture of the proposed Scratchpad

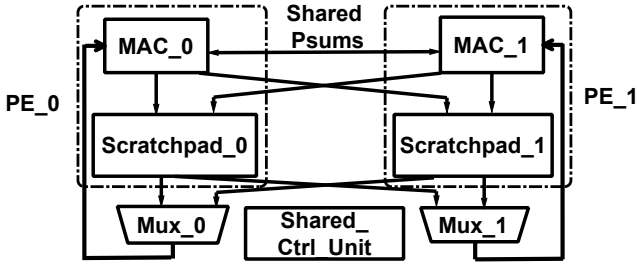


Figure 4.6: The internal architecture of the proposed Twin-PE.

MAC unit has a control unit that coordinates the computation process within that MAC unit. When the computation of a processing cycle is over, the control unit in each MAC unit notifies this issue to the main control unit through done signal.

### 4.3.2 Scratchpad Unit Specifications

Based on our dataflow, the loaded Ifmap and filter channels from DRAM are stored in scratchpad units. This unit is shown in Figure 4.5. Each scratchpad unit contains twelve 8×16b Ifmap scratchpads, four 4×9b zero scratchpads, three 32×15b filter scratchpads, and a 16×9b sign scratchpad. All the scratchpads are in dual port register file format. Twelve 8×16b Ifmap scratchpads are divided to two groups of six 8×16b scratchpads as a ping-pong buffer to provide the chance of reusing data locally [58]. Each group of six 8×16b scratchpads is divided to two groups of three 8×16b. Each group of three 8×16b Ifmap scratchpads is used to store the Ifmap rows of even or

odd channels. Each 4×9b zero scratchpad is used to store the zero flags regarding even or odd channels stored in a group of three 8×16b Ifmap scratchpads. The Ifmap and zero scratchpads that are loaded with even or odd channels, are distinguished with different colors in Figure 4.5. Each PE has an ID signal which equips the main control unit with support of different mapping format needed for mapping different layers of DNNs including depth-wise and 1×1 convolutional layers. In fact, the control unit of scratchpad unit, only writes the data into scratchpads when the associated ID with each chunk of data matches with the given ID to each PE. The control unit also provides valid signals for each stored Ifmap row, which are used by the address decoder units inside MAC units. The bit width of Ifmap and filter ports is 48 bits. Each port contains three elements.

### 4.3.3 Twin-PE

The internal architecture of the proposed Twin-PE is shown in Figure 4.6. This unit contains two MAC units, two scratchpad units and their corresponding multiplexers, and a shared control unit. Each single PE contains a MAC and a scratchpad unit which is distinguished with dashed lines in Figure 4.6. The output ports of the scratchpad units are connected to MAC units through multiplexers. The shared control unit coordinates the sharing process between the PEs. This unit shares scratchpad0 between PE0 and PE1 when PE1 is idle or shares scratchpad1 between PE0 and PE1 when PE0 is idle. It starts the sharing process based on the feedback that it gets from the main control unit. For example if PE1 is idle, the main control unit notifies this issue to the shared control unit. Then the shared control unit notifies the sctracthpad0 unit to restrict MAC0 unit from performing the computations of odd rows. The sctracthpad0 restricts the MAC0 unit from performing the computations of odd rows using the valid signals considered for each row. The shared control unit also connects the sctracthpad0 to MAC1 through the multiplexer to perform the computations of odd Ifmap and filter rows stored in sctracthpad0 simultaneously with MAC0.

### 4.3.4 Computation Engine Architecture and NoC Specifications

Figure 4.7 shows the general architecture of our design. Our computation engine contains sixteen PE clusters, four column GBs and four row GBs. Each PE cluster is a 3×3 matrix of PEs. Each column of the PEs computes a different output feature. In our design, each Ifmap and filter row contains three elements. All the filter and Ifmap rows with more than three elements are

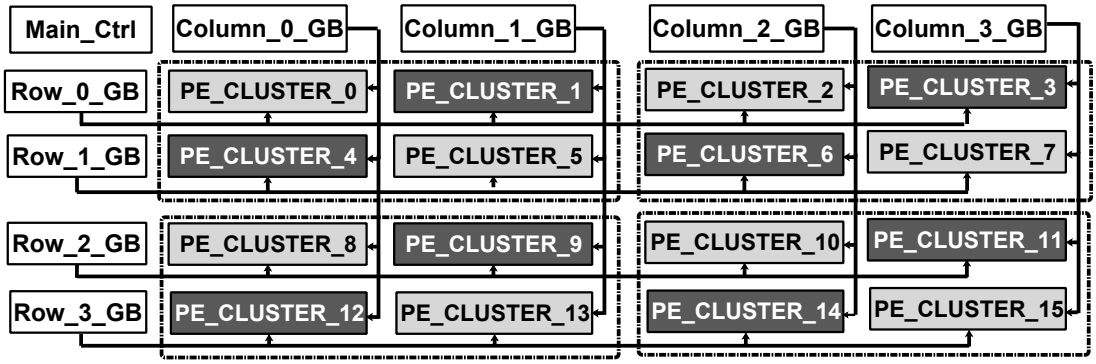


Figure 4.7: The general architecture of the proposed accelerator.

divided into rows containing three elements and zero padding is applied whenever needed. Each column GB contains two groups of three 6kB SRAM buffers for keeping the elements of Ifmap rows, four 1.5kB SRAM buffers for keeping the PSum values. In addition, four 1.2kB SRAM buffers for keeping the sign of the loaded filter weights in each row GB are shared among the column GBs. Each row GB contains three 6kB SRAM buffers for keeping the elements of each row of the filters. Each 6kB SRAM buffer in each row GB is considered for storing one element of each filter row while performing the computations of convolutional or depth-wise convolutional layers. However, this buffer in each row GB is considered for storing all the elements of one filter while performing the computations of 1×1 convolutional layers. The Ifmap and filter rows in each PE cluster are distributed using Row Stationary (RS) dataflow format introduced in [1] which comes with diagonal reuse of the Ifmaps in a 3×3 matrix of PEs while performing the computations of convolutional or depth-wise convolutional layers. In addition, An ID from 0 to 8 is given to each PE of each cluster, starting from the first PE in the first row to the third PE in the third row.

There are also four small control units in each column GB that coordinate the write or read processes into or from the PSum buffers. These units also coordinate the computations regarding the pooling layers. To perform Max-Pooling or Average-Pooling with a size [2,2], when the last output feature among four neighbor output features is calculated, these control units perform the required computations regarding the Max-Pooling or Average-Pooling layers before writing the final output feature into the PSum buffers. In addition, to shortcut or bypass the outputs between different layers, these units read the computed output features form the former layers that are pre-loaded in the PSum buffers and accumulate it with the recent computed output feature by the

PEs.

## 4.4 Proposed Dataflow

### 4.4.1 Sharing The Scratchpads of the PEs of Different Clusters

As shown in Figure 4.7, the PE clusters in our design are divided into four different groups. Each group has four clusters, which are distinguished with dashed lines. To get the optimum speedup, the PEs in clusters with same color in each group share their scratchpads. The PEs from different clusters that the summation of their given ID equals eight, cooperate with each other. Furthermore, cross-wise sharing the scratchpads in each group provides the chance for both clusters in one column and both clusters in one row of each group to cooperate with each other. In the general architecture shown in Figure 4.7, Row0 and Row1, Row2 and Row3, Column0 and Column1, and Column2 and Column3 can cooperate with each other.

Speedup in a group of PE clusters leads to acceleration of the whole computation engine if all the groups in the computation engine have similar or close behavior while performing the computations of a specific number of Ifmap and filter channels. To provide the chance of further speedup for all the groups using the available slack time caused by performing the computations of a few filters concurrently, the filters are sorted in an offline phase. First, pairs of filters that root-mean square of difference of their elements are smallest, and their number of negative elements are closest, are found. This simply means, there is a higher probability that the computation delay of the output features regarding these two filters are close compared to regular dataflow. Then two pairs of filters are chosen to be loaded into the computation engine: one pair of filters with the most number of negative weights and one pair of filters with least number of negative weights. Filters with the most number of negative weights are loaded into even rows and filters with least number of negative weights are loaded in odd rows of the computation engine. In addition, since neighbor Ifmap rows typically share close values [20], to provide the chance of further speedup for all the groups using the slack time caused by zero values, Ifmap rows are chosen from two different parts of the Ifmap channels. Then even columns of the computation engine are loaded with neighbor Ifmap rows of the first part and odd columns of the computation engine are loaded with neighbor Ifmap rows of the second part.

### 4.4.2 Negative Output Detection

As mentioned in Section 4.2, the Ifmap channels that their corresponding filter channels have the least number of negative weights are loaded first. The partial sums of the filters in each column are stored in the partial sum SRAM buffers available in their corresponding column GB. In each PE, the process of reordering the computations is the same as [58].

## 4.5 Results and Discussion

### 4.5.1 Implementation

Since the size of the computation engine and the used NoC format plays a key role in a system performance [23], to have a fair comparison, we created another version of the shown architecture in Figure 4.7 without sharing the scratchpads. This version is called as reference design in this cahpter. In fact, the reference design only supports computation-pruning techniques. A cycle-accurate simulator is developed to model both architectures. Both designs are implemented in VHDL and are synthesized using SYNOPSYS Design Compiler and TSMC 65nm technology library. ARM Artisan 65nm SRAM and register file generator are used for creating the global buffer and scratchpads. The clock frequency is 200MHz and the core voltage is 1.1V for both designs.

### 4.5.2 Hardware Performance Evaluation

Three popular classifiers are used for evaluating the proposed architecture, namely AlexNet [55], VGG16 [15], and MobileNet V2 [60]. The Batch size while doing inference for all the benchmark network is one. The achieved speedups for different layers of benchmark networks, caused by equipping the reference design with the proposed technique in section 4.2 are shown in Figure 4.8, Figure 4.9, and Figure 4.10. Due to the large number of layers of MobileNet V2 only the achieved speedup for a few layers of this network are shown in Figure 4.10. In each processing cycle four channels of the Ifmap rows are distributed among the PEs in each PE cluster. The average speedup while running AlexNet, VGG16, and MobileNet V2 are $\times$1.22, $\times$1.27, and $\times$1.22 respectively. The throughput and energy consumption comparison of both design are shown in Table 4.1. The average

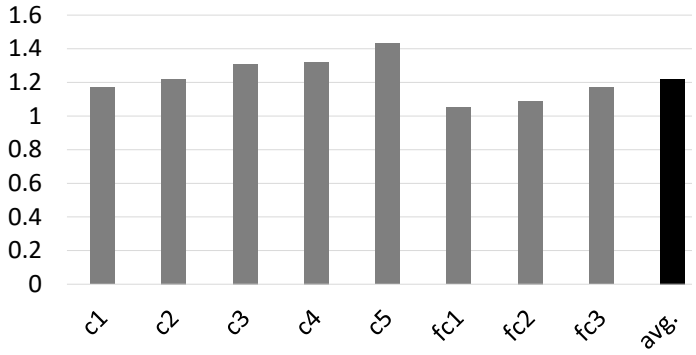


Figure 4.8: Speedup of layers of AlexNet by sharing the scratchpads.

power consumption of reference design and our proposed architecture is 287.6 mW and 300.3 mW respectively. The computation engines account for 72.2% and 73.2% of the consumed power in the reference design and our proposed architecture respectively. The average energy consumption of reference design and our proposed architecture for one inference while running AlexNet are 11.8 mJ and 9.7 mJ, while running VGG16 are 199.4 mJ and 156.9 mJ, and while running MobileNet V2 are 2.9 mJ and 2.5 mJ respectively. The area overhead of applying our method compared to consumed area of the logic of the reference design is 13.53% and to the whole reference design 1.68%. For both designs, 11.6%, 9.8%, and 8.9% of computations of different layers of AlexNet, VGG16, and Mobilenet V2 respectively and on average 10.1% of the computations of different layers of benchmark DNNs are skipped. As shown in Figure 4.10, it is possible to accelerate the computations of different layers including depth-wise and 1×1 convolutional layers based on the existing slack time while performing the computations of each layer and by sharing scratchpads of the PEs. The proposed technique is also applied into the computation engine of [58] and an average speedup of ×1.12 is achieved while running AlexNet and VGG16. The attained speedup of different layers in [58] caused by equipping the design with the proposed architecture in this work is affected by the low provided bandwidth of the used NoC format compared to used architecture in this cahpter.

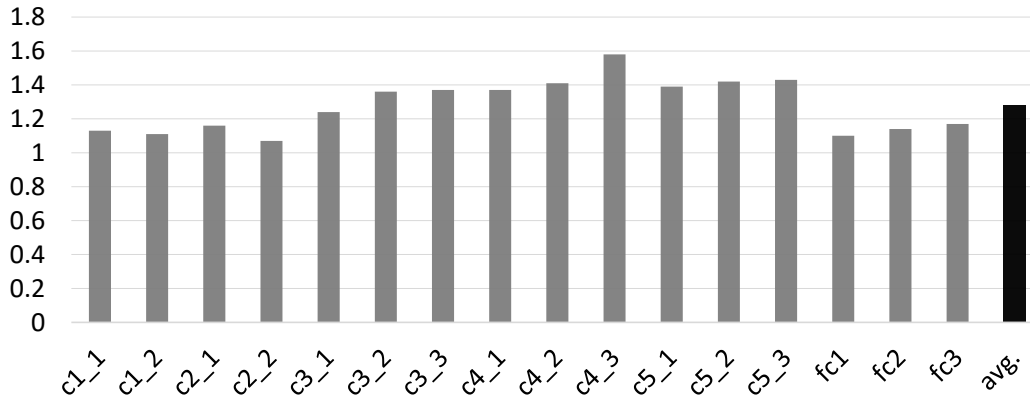


Figure 4.9: Speedup of layers of VGG16 by sharing the scratchpads.

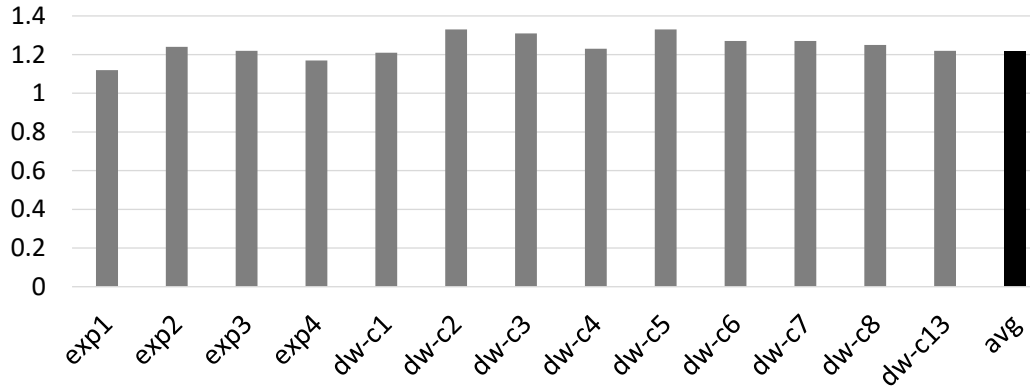


Figure 4.10: Speedup of some layers of MobileNet V2 by sharing the scratchpads.

Table 4.1: Comparison of throughput and energy consumption

| Hardware Specs. | Reference Design | | | Reference Design+Sharing | | |
|---|---|---|---|---|---|---|
| Area (logic only) | 1726K (NAND2) | | | 1959K (NAND2) | | |
| SRAM \| Reg. file | 244.8kB \| 57.3kB | | | 244.8kB \| 57.3kB | | |
| PEs | 144 | | | 144 | | |
| PRF Metric | AlexNet | VGG16 | MobileNetV2 | AlexNet | VGG16 | MobileNetV2 |
| Inference/S | 24.2 | 1.5 | 97.71 | 29.54 | 1.91 | 119.53 |
| Inference/J | 84.2 | 5.2 | 339.71 | 98.33 | 6.37 | 397.9 |

# 5. Pruning the Effectual Computations in DNNs

Integrating Deep Neural Networks (DNNs) into the Internet of Thing (IoT) devices could result in the emergence of complex sensing and recognition tasks that support a new era of human interactions with surrounding environments. However, DNNs are power-hungry, performing billions of computations in terms of one inference. Spatial DNN accelerators in principle can support computation-pruning techniques compared to other common architectures such as systolic arrays. Energy-efficient DNN accelerators skip bit-wise or word-wise sparsity in the input feature maps (ifmaps) and filter weights which means ineffectual computations are skipped. However, there is still room for pruning the effectual computations without reducing the accuracy of DNNs. In this chapter, A novel real-time architecture and dataflow is proposed by decomposing multiplications down to the bit level and pruning identical computations in spatial designs while running benchmark networks. The proposed architecture prunes identical computations by identifying identical bit values available in both ifmaps and filter weights without changing the accuracy of benchmark networks.

To be more specific, Section 5.1 provides a brief introduction, Section 5.2 shows the effectual computation reduction potential when performing the computations in bit level. Section 5.3 explains the proposed computation techniques. The proposed architecture is described in Section 5.4.

---

A major portion of this chapter is published in M. Asadikouhanjani, H. Zhang, L. Gopalakrishnan, H.-J. Lee and S.-B. Ko, "A Real-Time Architecture for Pruning the Effectual Computations in Deep Neural Networks," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 2030-2041, May 2021, doi: 10.1109/TCSI.2021.3060945

M.A conceived of and designed the proposed accelerator. M.A carried out all the simulations, implementations, and analysis. M.A wrote the manuscript in consultation with S.K., H.Z., H.L., and L.G.

Section 5.6 introduces our dataflow. Section 5.7 compares the performance of this work to the reference design in terms of throughput and energy consumption.

## 5.1 Introduction

IoT devices come with benefits like efficient resource utilization, minimizing human effort, and saving time. Applying deep learning can enable IoT devices to intelligently react to both user and environmental events. However, relying on battery power comes with its own set of challenges [61]. In addition, as IoT is developing rapidly, the security of these devices has become extremely prominent. Designers use countermeasure circuits beside the circuits that perform the main function of the device to protect these circuits in the implementation phase [62] which simply means securing the chip comes with more resources and energy consumption. Therefore, efficient implementation of DNNs is vital before integrating them into IoT devices.

Different DNN accelerators have been proposed for accelerating DNNs on various platforms until now [3, 59, 63–65]. Expensive data movements are a bottleneck for GPU, ASIC, and FPGA implementations. In [54], authors showed that performing the computations in memory results in remarkable energy efficiency. These types of accelerators try to avoid data movement between memory and processing engine. However, in-memory architectures suffer from utilizing Analog to Digital Converters or Digital to Analog Converters, which take a big area and power consumption percentage of the whole chip. In addition, their non-generic layouts are expensive to build since it is usually a mix of CMOS and other technologies.

The Network on Chip (NoC)-based DNN accelerator introduced in [23] comes with reducing off-chip memory accesses and enhancing run-time computational flexibility benefits. The authors showed that efficient Processing Element (PE) and NoC designs are both critically important to keep the performance of the whole design high in the NoC-based DNN accelerators. Memory access in [23] is reduced by 88% up to 99% compared to the conventional DNN accelerators. Eyeriss V2 [24] is an energy-efficient spatial DNN accelerator that exploits a flexible NoC which provides the required bandwidth of the computation engine while running various DNNs. There are also other works that focused on efficient data movement inside the computation engine [66, 67]. With the use of the proposed NoC types in the aforementioned works and data reuse techniques, it is

possible to decrease the number of memory accesses. However, skipping bit-wise sparsity is not explored in [23] and [24].

Authors in [68, 69] proposed algorithms to reduce the multiplication operations inside the computation engine. In [58], the authors of this chapter proposed a real-time architecture for early detection of negative output features in spatial DNN accelerators. The computations are reordered using sign tables inside and outside of the computation engine. The sign table which is outside the computations engine contains the sign of all the weights of the loaded filters. Each of the sign tables inside the PEs of the computation engine, contains the sign of a few specific filter rows. With the use of the sign table outside the computation engine, there is no need to postpone performing the computations until all the corresponding ifmap rows of the preloaded filter rows are loaded inside the computation engine. This method decreases the number of Multiply and Accumulate (MAC) operations and the number of memory accesses. This accelerator is built upon Eyeriss [1]. The used NoC format in [58], like Eyeriss cannot provide the needed bandwidth of the PEs which causes stalls in the computation engine.

Authors in [70], proposed the design of a C-programmable application-specific instruction set processor (ASIP) and achieved an energy efficiency of $\times 5$ compared to their closest competitor design. This result was achieved by compressing the benchmark DNN and using dynamic voltage scaling. Their design also supports skipping sparsity. However, as they mentioned, their proposed techniques are suitable for applications that require less accurate computations. In [27], the authors presented a method for transparently identifying ineffectual computations during inference with deep learning models. Specifically, by decomposing multiplications down to the bit level. This work targets bit sparsity, that is zero bits since processing zero bits in a MAC operation does not affect the outcome. As shown in [27], the number of multiplications during inference can be potentially reduced by at least $40\times$ across a wide selection of neural networks. Furthermore, the authors in [27] proposed a novel PE that is $5\times$ smaller and much more energy-efficient than the most identical competitive works [3]. The proposed architecture in [28] only skips computation cycles for zero-valued weights and the proposed architecture in [29] only skips computation cycles for zero-valued ifmaps. However, in these type of accelerators, the focus is on skipping the ineffectual computations [30, 31]. In this chapter, our focus is on improving the energy efficiency

of the accelerators that decompose the computations down to bit-level and skip sparsity. We built upon [27] and show that there is still room for pruning the effectual computations. The contributions of this chapter are:

1. A novel real-time architecture is proposed for the spatial DNN accelerators which not only skips the ineffectual computations but also it prunes the effectual computations during inference while maintaining the accuracy.

2. A speedup of ×1.17 is achieved while performing the computations of convolutional layers with different filter sizes and expansion and projection layers compared to the reference design.

3. A novel dataflow is proposed for the layers with a limited chance of reuse of the ifmaps and filter weights such as fully connected and depth-wise convolutional layers. A speedup of ×2.45 is achieved by using our proposed dataflow while performing the computations of depth-wise convolutional and fully connected layers compared to the reference design.

4. In total, an energy efficiency per inference of ×1.21 is achieved with only 24.75% and 7.21% area overhead compared to the consumed area of the logic and the whole reference design respectively. This improvement comes by pruning the identical computations among the PEs compared to the reference design.

## 5.2 Identical Effectual Computations

Most of the proposed energy-efficient accelerators until now focused on skipping word-wise or bit-wise sparsity available in the ifmap and filter weights. However, this section explains why it is possible to optimize these accelerators further by targeting identical computations when the ifmaps and filter weights are decomposed down to bit level. Identical computations refer to the MAC operations that produce equal results. As an example, Figure 5.1 shows the identical MAC operations in a 2×1 spatial computation engine while MAC operations are decomposed down to bit-level. In this example, the size of the ifmap and filter rows is 1×1. The considered bit-width of the ifmap and filter weights is 16b. To reuse the loaded ifmap locally inside the computation engine as much as possible, PE1 and PE2 share the same ifmap. Each PE performs the multiplication of

the radix-4 Booth-encoded format of the loaded ifmap and filter weight. In each clock cycle, the MAC unit inside each PE only accepts a non-zero Booth-encoded element of the ifmap and filter weight. PE1 performs the computations of filter 1 and PE2 performs the computations of filter 2. Two corresponding elements of the weights of filters 1 and 2 are equal. These identical elements are distinguished with the same color in Figure 5.1 (a). Accordingly, the result of the MAC operations of these elements would be equal.

Figure 5.1 (b) shows the dataflow of the PEs inside the computation engine shown in Figure 5.1 (a). Each PE performs 12 MAC operations. The total number of MAC operations without pruning identical computations is 24 inside the computation engine. As shown in Section 5.3, it is possible to perform identical MAC operations only once instead of repeating these MAC operations in both of the PEs. Since 8 MAC operations are repeated in each PE, by performing the identical MAC operations only once, the total number of MAC operations decreases to 16 inside the computation engine. The same thing is true for identical elements of Booth-encoded format of the ifmaps when performing the computations of a filter and a few parts of the image simultaneously. In this work, an architecture is proposed which can efficiently prune identical MAC operations in the state-of-the-art DNN accelerators [27].

### 5.2.1 Identical bit values among filter weights

Like [27], our proposed design performs the computations of a few filters at the same time to reuse the ifmaps locally. However, the computation engine prunes the identical computations by identifying identical bit values of the corresponding loaded weights of different filters. In our design, the weights of all the filters are encoded using the radix-4 Booth format. The feasibility of pruning the identical computations by identifying identical bit values of the filter weights is shown by the following example: The size of each filter in the 9th convolutional layer in VGG16 is 3×3×512. There are 512 filters in the 9th convolutional layer of VGG16 to compute 512 output channels. Figure 5.2 (a), (b), (c) and (d) show the distribution of the weights of the filters 17, 244, 245, and 421 of the 9th convolutional layer of VGG16 [15] respectively. As discussed, filters 17, 244, 245, and 421 compute the 17th, 244th, 245th, and 421th output channel of the 9th convolutional layer of VGG16 respectively. The considered bit-width for the filter weights is 16b. It is obvious

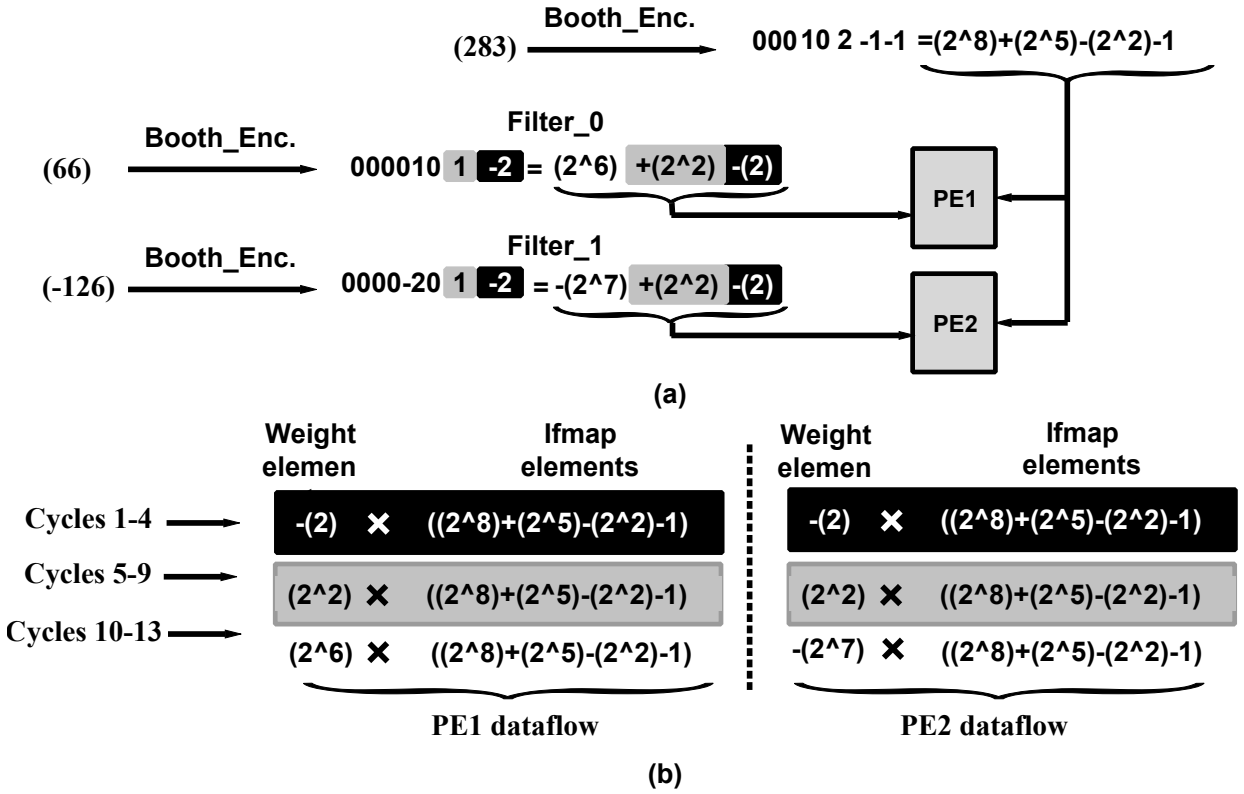


Figure 5.1: The identical MAC operations by identifying (a) identical elements of the corresponding Booth-encoded weights and (b) their associated identical dataflow in a 2×1 spatial computation engine.

that these filters approximately follow a Gaussian alike distribution. As shown in this figure, the majority of the weights of all the filters are distributed around zero which means many of these weights have close values. Accordingly, there is a high chance for the Booth-encoded format of the high nibble of these close values to share identical elements. In addition, the value of each element of each encoded weight could only be a fixed signed number between -2 and 2. This means there is a good chance for the corresponding encoded weights of different filters to share identical elements. For example, for all the filters of the 9th convolutional layer of VGG16 [15], it is possible to group the filters into pairs of filters that on average in each pair, regardless of zero values, 8.1% of the elements of different encoded weights have identical value.

As shown in Section 5.7, the same thing is true with different layer types of the benchmark networks. In what follows the offline sorting process of the different filters of each layer is explained

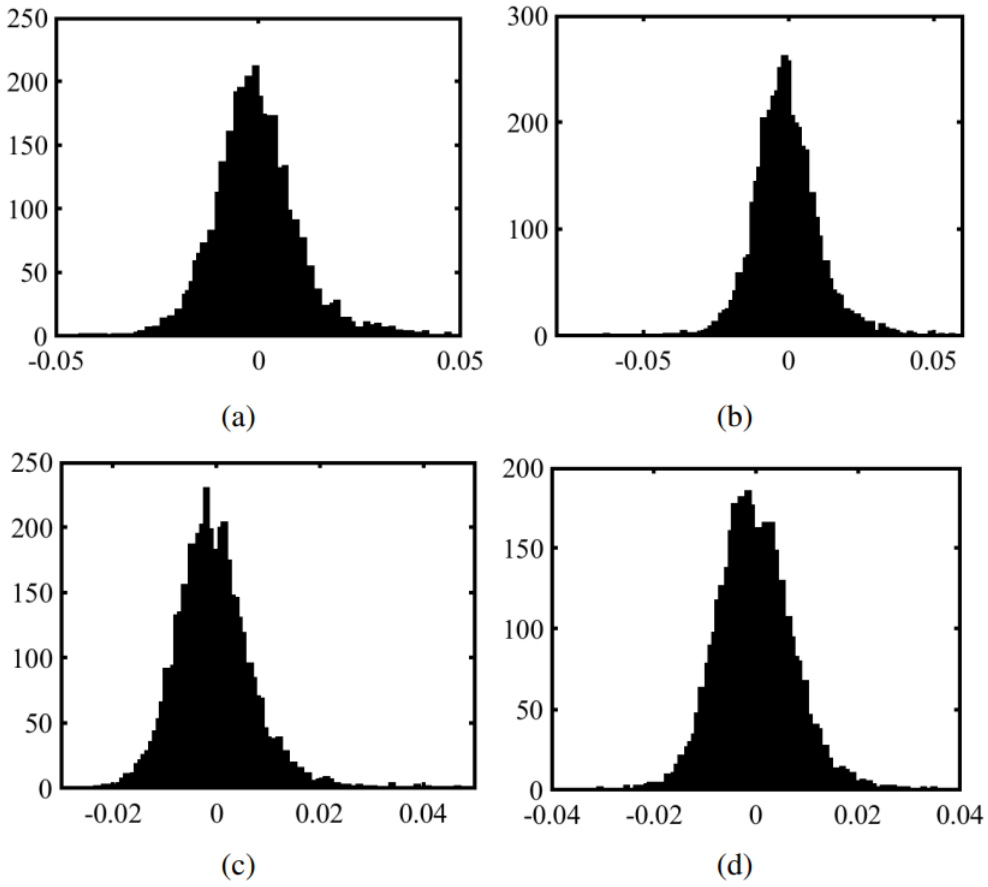


Figure 5.2: Distribution of the weights of filters (a) 17 (b) 244 (c) 245 (d) 421 of 9th convolutional layer of VGG16.

in more detail. Like [27], it is possible to load a few filters simultaneously to reuse the loaded ifmaps locally in our design. Since 16b is considered as the bit width of filter weights, the Booth-encoded format of each filter weight has 8 elements. Based on our proposed computation technique detailed in Section 5.3, the energy efficiency of our method comes from pruning the identical computations of the considered pairs of filters. Therefore, the energy efficiency of our design would be different for different ways of selecting these pairs of filters. To achieve the optimum energy efficiency while performing the computations of different filter pairs of each layer of the benchmark networks, the filter pairs must be carefully selected. In more detail, these pairs are found so that the average total number of similar elements of the corresponding weights of all the selected filter pairs is the highest. Using this sorting method, it is possible to get more energy efficiency compared to regular dataflow. Finding the optimum pairs of filters is an offline process since the weights are available

after training the DNN. For example, for the filters shown in Figure 5.2, the percentage of the identical elements of the corresponding encoded weights of filters 244 and 17 is 8.5% more than that of filters 244 and 245. In addition, the percentage of the identical elements of the corresponding encoded weights of filters 245 and 421 is 1.5% more than that of filters 245 and 244. It indicates that performing the computations based on the regular dataflow might not provide the chance to prune effectual computations efficiently. However, performing the computations of filters 244 and 17 of the 9th convolution of VGG16 provides the chance of pruning identical computations more. The same thing is true for filters 245 and 421. Filters 244 and 17 and filters 245 and 421 share 9% and 8% of their whole encoded elements respectively. As mentioned above, these percentages are regardless of zero values.

### 5.2.2 Identical bit values among ifmaps

In [71], the authors proposed a value prediction method that exploits the spatial correlation of the ifmaps inherent in DNNs. They showed that neighbor output features share close values in different layers of DNNs. Accordingly, there is a high chance for the Booth-encoded format of these close values to share a few elements. As explained in Section 5.4 of this chapter, our proposed architecture is a spatial design that performs the computations of a few neighbor Ifmpas concurrently. Furthermore, the results in Section 5.7 confirm that there is still room to prune the effectual computations of the neighbor ifmaps. The amount of identical computations of identical bit values of the ifmaps in low depth layers is higher than that of high depth layers because of the amount of sparsity among the ifmaps increases in high depth layers.

## 5.3 Proposed Computation Technique

To introduce the effectual computation pruning technique, a common 2×2 version of the architecture used in [3,58,59], is chosen and explored. This architecture which provides the chance of reuse of the ifmaps and filter weights locally, is shown in Figure 5.3. The chosen architecture is equipped with horizontal and vertical 1D multi-cast NoC types. Less needed bandwidth and high data reuse are the main features of this type of NoC compared to other types such as uni-cast and 1D systolic. In our design, the Booth-encoded format of each ifmap and filter weight has 8

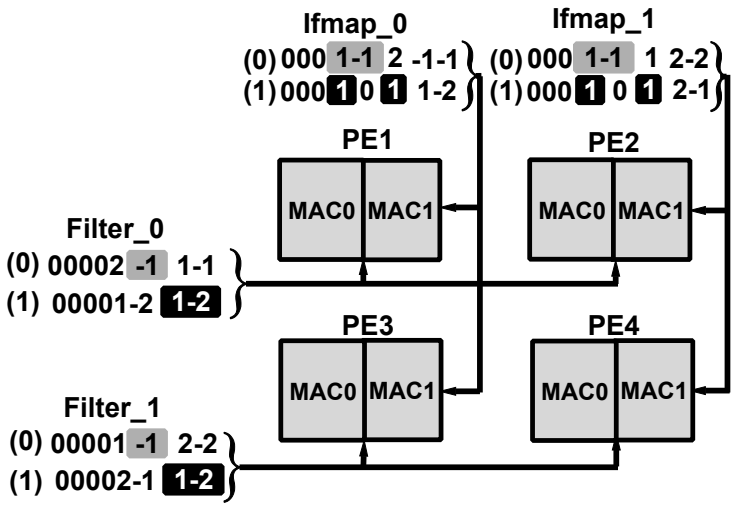


Figure 5.3: A 2×2 spatial computation engine that performs the computations of Booth-encoded ifmaps and filter weights using regular PEs.

elements since the considered bit-width for the ifmaps and filter weights is 16b. In this example, the size of each of ifmap and filter row is 1×1. Furthermore, two rows of two different filters and parts of the input image are loaded inside the computation engine. In addition, each PE has two MAC units. Each of these units performs the computations of each row.

Like [27], the computation delay of each MAC unit inside each PE, is equal to the number of non-zero elements of the loaded filter rows times the number of non-zero elements of the loaded ifmap rows inside that MAC unit. Accordingly, the computation delay of each PE is equal to the maximum computation delay of the MAC units inside that PE. For example, the computation delay of MAC0 and MAC1 in PE1 are 20 and 16 cycles respectively. Therefore, the computation delay of this PE is 20 cycles. The identical elements of the corresponding rows of the two loaded filters are distinguished with the same color in Figure 5.3. The same thing is true with the corresponding rows of the two loaded parts of the input image.

As shown in Figure 5.3, identical computations are performed by the PEs located in different rows of the computation engine. These computations are identical because of the existing identical elements in the corresponding rows of the loaded filters. For example, both PE1 and PE3 that share the same ifmap rows, perform a number of identical computations because of the identical elements inside the corresponding rows of the two loaded filters inside these PEs. Likewise, the existence

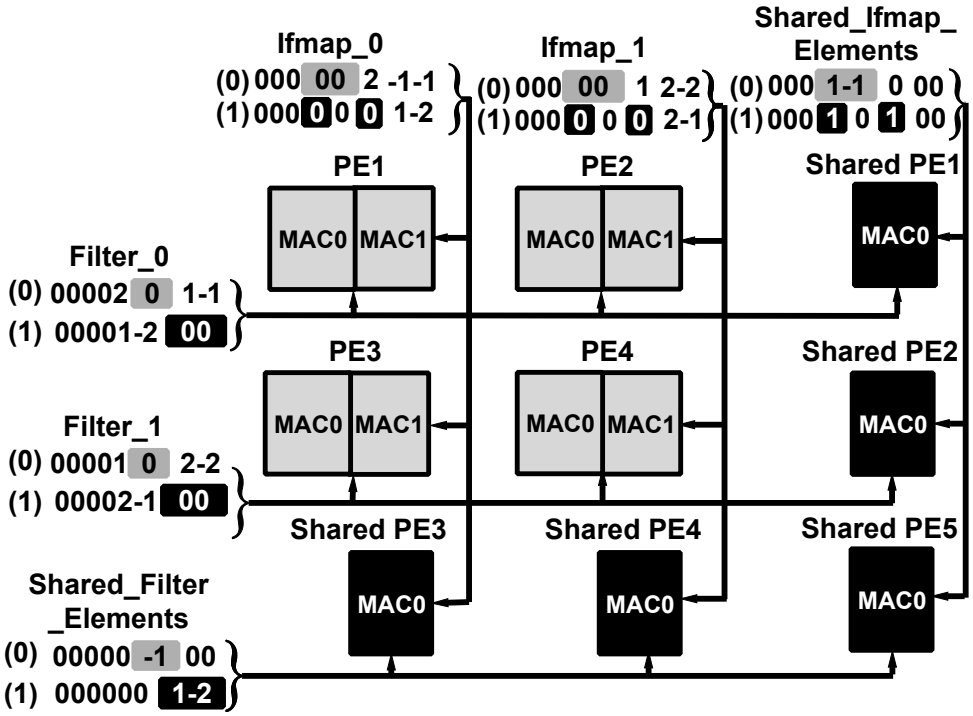


Figure 5.4: A 3×3 spatial computation engine that prunes the computations by identifying identical bit values in ifmaps and weights only once using regular and shared PEs.

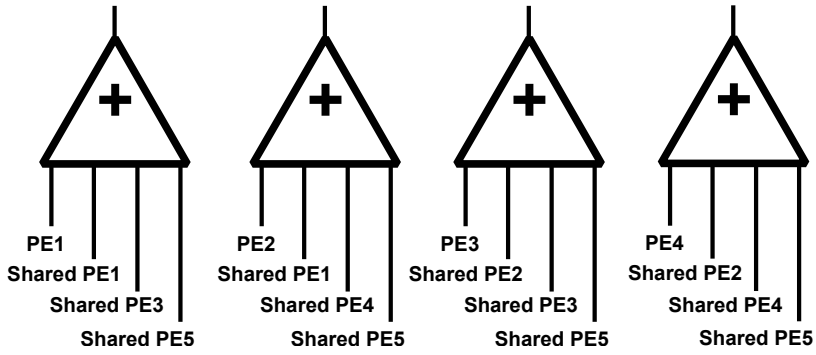


Figure 5.5: Adder trees that are considered to calculate the final partial sums out of the computed partial sums by both regular ans shared PEs in a 3×3 spatial computation engine.

of identical elements in the corresponding rows of the two loaded parts of the input image causes the PEs that are located in different columns of the computation engine to repeat a few identical computations.

To prune the identical computations based on our proposed computation technique, the archi-

tecture shown in Figure 5.3 is changed to the architecture shown in Figure 5.4. In this figure, identical elements in the corresponding rows of the two loaded filters and in the corresponding rows of the two loaded parts of the input image are replaced with zero for regular PEs. As indicated in Figure 5.4, compared to the design shown in Figure 5.4, a few extra PEs which are named as shared PEs, are added to the design to perform the computations of the identical elements. In fact, in our proposed architecture, identical computations are performed only once by the shared PEs that are distinguished by black color. In this example, the shared PEs contain only one MAC unit. Shared PE1 and shared PE2 are considered to perform the computations of identical elements of the corresponding rows of the two loaded parts of the input image. Shared PE3 and shared PE4 are considered to perform the computations of identical elements of the corresponding rows of the two loaded filters. Shared PE5 performs the computations of both identical elements of the corresponding rows of the two loaded parts of the input image and identical elements of the corresponding rows of the two loaded filters. In this architecture, when different PEs are done with their given tasks, they feed the considered adder trees with their computed partial sums to compute the final partial sum. These adder trees and their relative inputs are shown in Figure 5.5.

The computation delay and the number of MAC operations of the PEs of both architectures are reported in Table 5.1. For example, the number of MAC operations performed in PE1 before and after the pruning identical computations are 36 and 13 respectively. In addition, the computation delay of this PE before and after pruning identical computations are 20 and 9 respectively. As there is only one MAC unit in each shared PE, the computation delay of each shared PE is equal to the processing delay of performing the computations for the first loaded filter and ifmap row plus the processing delay of performing the computations for the second loaded filter and ifmap row in that shared PE. When all the regular and shared PEs are done with their given tasks, the main control unit calculates the total partial sums using the adder trees in one clock cycle. The reported total MAC operations when identical computations are pruned is by considering the accumulations which are performed inside each adder tree shown in Figure 5.5. As it is clear, by pruning identical computations, the computation delay of the whole computation engine is decreased by $\times 1.8$. Furthermore, the number of MAC operations is decreased by $\times 1.38$.

Our proposed computation technique comes with a minimal area overhead which is mostly

Table 5.1: Comparison of computation delay and number of MAC operations before and after pruning identical computations

| PE Specs. | Computation Delay | | Number of MAC Operations | |
|---|---|---|---|---|
| Status | before pruning | after pruning | before pruning | after pruning |
| PE1 | 20 | 9 | 36 | 13 |
| PE2 | 20 | 9 | 36 | 13 |
| PE3 | 20 | 9 | 36 | 13 |
| PE4 | 20 | 9 | 36 | 13 |
| Shared PE1 | - | 10 | - | 10 |
| Shared PE2 | - | 10 | - | 10 |
| Shared PE3 | - | 7 | - | 7 |
| Shared PE4 | - | 7 | - | 7 |
| Shared PE5 | - | 6 | - | 6 |
| Adder trees | - | 1 | - | 12 |
| Total | 20 | 11 | 144 | 104 |

because of the added shared PEs. The internal architecture of the PEs in our design is identical to the proposed PE in [27]. This PE forgoes multipliers and shifters, expensive components, and instead utilizes narrow adders to perform the multiplication operations. Therefore, this PE is very optimized in terms of resource and power consumption. Furthermore, based on our evaluation of different layers of benchmark networks, since the number of MAC operations of identical elements is lower, the number of MAC units in the shared PEs could be lower than regular PEs. As explained in Section 5.7, the size of each shared PE is almost $\times 4$ smaller than each regular PE. Accordingly, the overall area overhead of our proposed architecture compared to reference design is only 7.21%. In addition, in the given example, it is possible to remove shared PE5 and perform the computation of identical encoded weights and ifmaps by shared PE1 and shared PE2. However, the speedup of our proposed computation technique in this example decreases to $\times 1.2$. In fact, the speedup decreases sharply since PE5 performs the computations of both identical ifmaps and filter weights.

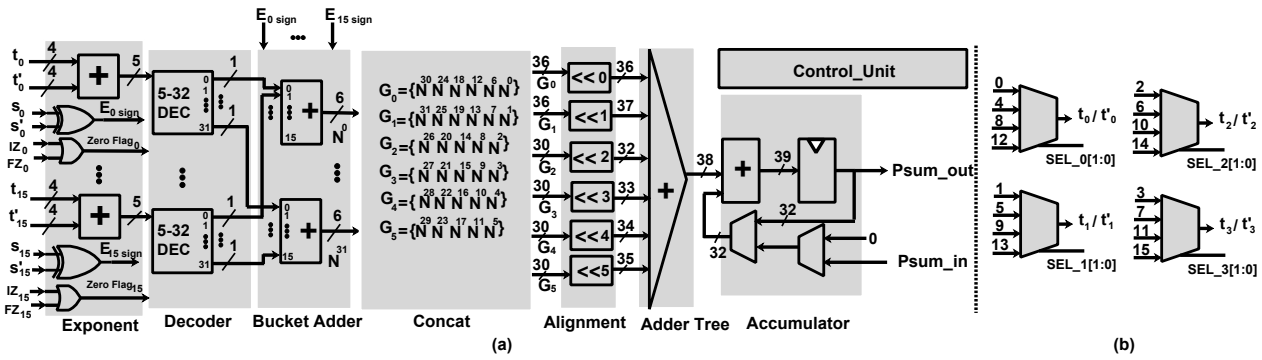


Figure 5.6: The internal architecture of the (a) regular PEs and the (b) considered multiplexers in the shared PEs for selecting the corresponding ifmaps or filter weights

## 5.4 The Architectures of Regular and Shared PEs

The proposed PE in [27], calculates the product of a weight $W = (Wterms)$ and an input feature map $I = (Iterms)$ where each element is a $(sign, magnitude) = (s_i, t_i)$ as shown in (1).

$$W \times I = \sum_{\forall (s,t) \in Wterms} (-1)^s 2^t \times \sum_{\forall (s',t') \in Iterms} (-1)^{s'} 2^{t'} \tag{5.1}$$

That is, instead of processing the full $W \times I$ product in a single cycle, each processing lane inside each PE processes each product of a single $t'$ element of the ifmap $I$ and a single $t$ element of the weight $W$ individually. Since these terms are powers of two so will be their product. Accordingly, the processing lanes inside each PE can first add the corresponding exponents $t' + t$. The same approach can be used when processing 16 weight and activation pairs in parallel. In [27], each PE can perform the multiplication of 16 ifmaps and weights concurrently. The internal architecture of used PE is explained in what follows:

### 5.4.1 Regular PE Unit Specifications

The internal architecture of each regular 16b PE in our computation engine is shown in Figure 5.6 (a). There are 16 processing lanes in each PE. Each processing lane is composed of several units. These units are named as exponent, decoder, bucket adder, concat, and alignment. Exponent unit accepts 16 4-bit weight terms, $t_0, ..., t_{15}$ and their 16 corresponding sign bits, $s_0, ..., s_{15}$ and 16 corresponding zero flag bits, $IZ_0, ..., IZ_{15}$. Also, 16 4-bit activation terms, $t'_0, ..., t'_{15}$ and their 16

corresponding sign bits, $s'_0, ..., s'_{15}$, and 16 corresponding zero flag bits, $FZ_0, ..., FZ_{15}$. This unit calculates 16 terms pair products. Since all terms are powers of two, their products will also be powers of two. Accordingly, to multiply sixteen ifmaps by their corresponding weights, this unit adds their terms to generate the 4-bit exponents $(t_0 + t'_0), ..., (t_{15} + t'_{15})$ and uses 16 XOR gates to determine the signs of the products. Decoder unit calculates $2^{(t_i+t'_i)}$ via a 5b-to-32b decoder which converts the 5-bit exponent result $(t_i + t_i)'$ into its corresponding one-hot format, i.e., a 32-bit number with one "1" bit and 31 "0" bits. As explained above, each PE performs the computation of sixteen ifmaps and filter weights concurrently. When the zero flag bit relative to either a loaded ifmap or a filter weight is "1", it means that there is no need to perform any computations for that specific ifmap and filter weight.

Bucket adder unit accumulates the output of decoder unit into 32 buckets, $N^0, ..., N^{31}$ corresponding to the values of $2^0, 2^1, ..., 2^{31}$ as there are 31 powers of two. The signs of these numbers are also taken into account. Concat unit adds 32 6b counts that each corresponding to a different power of 2 between 0 and 31 in a very optimized way. like [27], the idea behind this unit is that it is possible to group $N^i$ where $i$ MOD 6 is equal. In fact, instead of shifting these 6b counts and accumulating them which is more expensive in terms of resource consumption. The outputs of concat unit are aligned by exploiting the relative weighting of the concatenated groups in the Alignment unit. The Adder tree in each PE accumulates the outputs of the alignment unit. The accumulator unit in a PE accumulates the current output of the adder tree with the former partial sum stored in that PE or with the computed partial sum of other PEs. In addition, the control unit in each PE coordinates the accumulation of the most recent computed partial sum with the previously computed partial sum or with the computed partial sum by other PEs to calculate the final output feature while performing the computations of different layer types of DNNs.

### 5.4.2 Shared PE Unit Specifications

The main difference between the internal architecture of the shared PEs compared to regular PEs is that there are only 4 processing lanes inside each shared PE. Therefore, the size of the input of the adders inside the bucket adder unit is decreased from 32 to 4 and the size of the output of these adders is decreased from 6 to 4. In shared PE, the concat unit adds 32 4b counts. Therefore,

the bit-width of $G_0$ and $G_1$ is decreased by 12 bits and the bit-width of $G_2, ..., G_5$ is decreased by 10 bits. Accordingly, the size of the adder tree is decreased in the shared PEs. However, the size of the accumulator in shared PEs is the same as regular PEs. In addition, as there are only 4 processing lanes inside each shared PE, 4 multiplexers are considered either for selecting ifmaps or filter weights in each shared PEs. In the shared PEs that perform the computations of shared ifmap elements, these multiplexers are used for selecting the corresponding weights. In the shared PEs that perform the computations of shared weight elements, these multiplexers are used for selecting the corresponding ifmaps. The size of these multiplexers are 4 to 1 to cover all the sixteen ifmaps or filter weights provided for the regular PEs. These multiplexers and the way that the sixteen ifmaps or filter weights are assigned to them are shown in Figure 5.6 (b). These multiplexers are discussed in more detail in the following section.

## 5.5 General Architecture of Our Proposed Accelerator

The general architecture of our proposed accelerator is shown in Figure 5.7. Our computation engine contains 16 regular PEs, 14 shared PEs, 4 column GBs, 4 row GBs, an ifmap and two filter distributor units. The regular PEs and the shared PEs are shaped into 5 columns and 6 rows. Each regular PE contains 16 processing lanes and each shared PE contains 4 processing lanes. Each column of the PEs computes 4 different output features of 4 different filters.

The main difference of the proposed computation engine to our reference design [27], is the added shared PEs and the added ifmap and filter distributor units. These units are distinguished with black color in Figure 5.7. As explained in Section 5.2, pairs of filters that their encoded weights have the most identical number of elements are founded in an offline phase. The proposed architecture shown in Figure 5.7, can perform the computations of two of those pairs concurrently. The main control unit loads the first pair of the filters into Row0 and Row1 GB and loads the second pair of the filters into Row2 and Row3 GB of the computation engine. The shared PEs 0 to 6 perform the pruned identical computations of identical bit values of the ifmap and filter weights that are loaded inside the regular PEs of Row0 and Row1 of the computation engine. In addition, shared PEs 7 to 13 perform the pruned identical computations of identical bit values of the ifmap and filter weights that are loaded inside the regular PEs of Row2 and Row3 of the computation

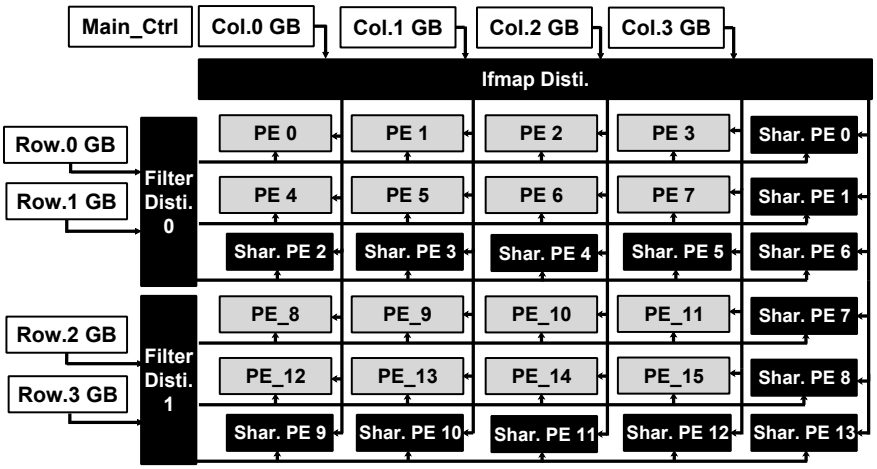


Figure 5.7: The general architecture of the proposed accelerator.

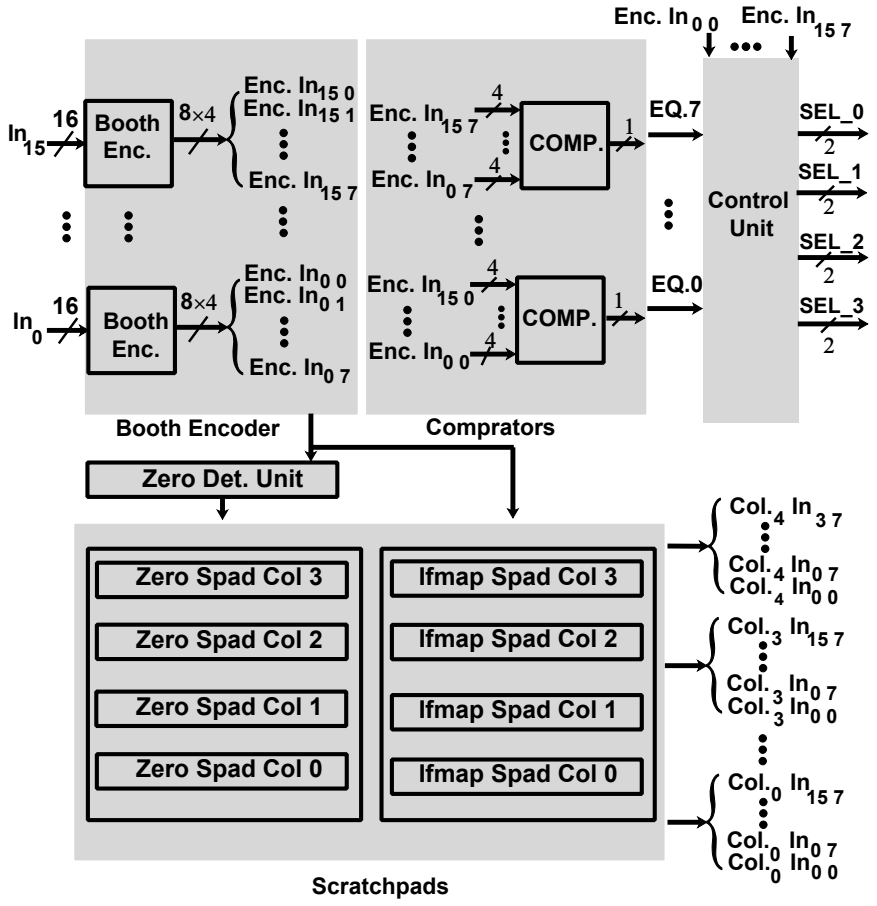


Figure 5.8: The internal architecture of the ifmap distributor unit.

engine.

### 5.5.1 Distributor Units

The internal architecture of the ifmap distributor unit is shown in Figure 5.8. There are 4 main sub-units inside each distributor unit. These sub-units are Booth encoder, comparator, scratchpad, and an internal control unit. The distributor unit can load up to 4 ifmaps concurrently from each of the Column GBs into the scratchpad unit. The Booth encoder unit encodes the loaded ifmaps. The comparator unit compares the elements of the corresponding encoded ifmaps of different GBs. The internal control in the distributor unit finds the identical elements of the corresponding encoded ifmaps. The zero detection unit, which is part of the internal control unit, detects and stores the zero flags of the elements of the encoded data.

In more detail, the ifmap distributor unit has four main jobs. The first job of this unit is Booth encoding the input data. The ifmap distributor unit, in each cycle, can read four ifmaps from the Column0 to Column3 GBs concurrently. This process continues based on the considered dataflow for performing the computations of each layer. Then, this unit feeds the PEs inside the computation engine with the Booth-encoded format of the loaded ifmaps. The second job of this unit is pruning identical computations. The ifmap distributor unit replaces identical elements of the encoded ifmaps with zero in regular PEs while performing the computations of the loaded pairs of filters. The ifmap distributor unit also loads these identical elements into the shared PEs located in the fifth column of the computation engine which is shown in Figure 5.7. Accordingly, our proposed computation engine prunes the effectual computations by identifying identical bit values of the loaded ifmaps and each loaded pair of filter rows only once. The third job is efficient distribution of the elements of the encoded ifmaps of layers with a limited chance of reusing data locally such as fully connected layers and depth-wise convolutional layers which is explained in detail in Section 5.6. The fourth job of this unit is generating zero flags relative to the elements of the encoded ifmaps. The outputs of the ifmap distributor unit are the encoded elements considered for different columns of the computation engine along with their corresponding zero flag bits. To present the internal architecture of the distributor unit of Figure 5.8 in a readable format, the zero flag bits that are also part of the outputs of this unit are not shown in this figure.

As shown in Figure 5.8, there are 4 multiplexer select signals that are also the outputs of the ifmap distributor unit. These signals are the inputs for the shared PEs that perform the computations

of the shared elements of the encoded ifmaps. The multiplexers inside the shared PEs are introduced in the former section. In more detail, for the shared PEs that perform the computations of identical elements of the encoded ifmaps, these multiplexer are specified to select the corresponding element of the encoded weights. For the shared PEs that perform the computations of identical elements of the encoded weights, these multiplexers are specified to select the corresponding corresponding element of the encoded ifmaps and weights.

The scratchpad unit in the ifmap distributor unit, contains 4 ifmap scratchpad and 4 zero flag scratchpad sub-units. Each ifmap scratchpad sub-unit contains 16 $8 \times 4$b+$8 \times 1$b registers to store the elements of the encoded ifmaps and their relative sign bits. Each zero flag scratchpad sub-unit that is considered for the PEs located in a specific column of the computation engine, contains 16 $8 \times 1$b registers. The zero flag registers in our ifmap and filter distributor units are considered to specify that the elements of the encoded ifmaps and filters are zero or not. This flag bits are used to skip sparsity available in the encoded format of the ifmaps and the weights. Furthermore, there are 16 $8 \times 1$b registers to store the equal flag bits of the available identical elements of the encoded corresponding ifmaps. The control unit inside the ifmap distributor unit uses these flags to prune the identical effectual computations.

The internal architecture of the filter distributor units is identical to ifmap distributor unit. The main difference of the filter distribution unit is that the identical elements of the corresponding encoded weights are pruned for each pair of filters separately. That is why two separate filter distributor units are considered. Therefore, the size of each filter distributor unit is almost half of the ifmap distributor unit. Furthermore, as explained above, the filter distributor units feed the select inputs of the multiplexers of the shared PEs that perform the computations of shared elements of the encoded weights.

The control unit inside each distributor unit coordinates the write and read data process into and from the mentioned scratchpads above. The main control unit starts each processing cycle by notifying the control unit inside the ifmap and filter distributor units to load their first stored non-zero element of all the encoded ifmaps or weights. Then, the main control unit notifies the internal control unit of the ifmap distributor unit to load all the remained non-zero elements of the encoded ifmaps into the PEs in each cycle. The main control unit repeats this process for each

non-zero remained element of the encoded weights inside the filter distributor units. As explained, the non-zero elements are chosen using the ifmap and filter zero flag bits.

### 5.5.2 Global Buffers

Each column GB contains 4 1.5kB SRAM buffers for keeping the elements of ifmap rows and a 1.5kB SRAM buffers for keeping the partial sum values. Each row GB contains 4 1.5kB SRAM buffers for keeping the elements of each row of the filters. There are 4 small adder trees in each column GB that are considered to accumulate the computed partial sums by specific shared and regular PEs to compute the final partial sums. Furthermore, 4 small control units in each column GB that coordinate the write or read processes into or from the partial sum buffers. These units also coordinate the computations of the pooling layers. To perform Max-Pooling or Average-Pooling with a size [2,2], when the last output feature among 4 neighbor output features is calculated, these control units perform the required computations of the Max-Pooling or Average-Pooling layers before writing the final output feature into the partial sum buffers. Also, to shortcut or bypass the outputs between different layers, these units read the computed output features from the former layers that are pre-loaded in the partial sum buffers and accumulate it with the fresh computed output feature by the PEs.

## 5.6 Proposed Dataflow

The mapping of different layer types of DNNs into the computation engine comes with two challenges. The first one is reusing data locally which mostly is relevant to the specification of each layer type. The second one is the optimal utilization of the available processing resources inside the computation engine. This simply means, to perform the associated computations of different layer types efficiently, each of these types needs to be mapped differently into the computation engine. In our design, the main control unit stores the specification of different layers of benchmark networks. This unit configures the ifmap and filter distributor units that feed the PEs of the computation engine based on those specifications. At the end of each processing cycle, when all the PEs are done with their given tasks, the main control unit notifies the control units inside the column GBs that the partial sum of the formerly given tasks are ready. Besides, the main control unit notifies

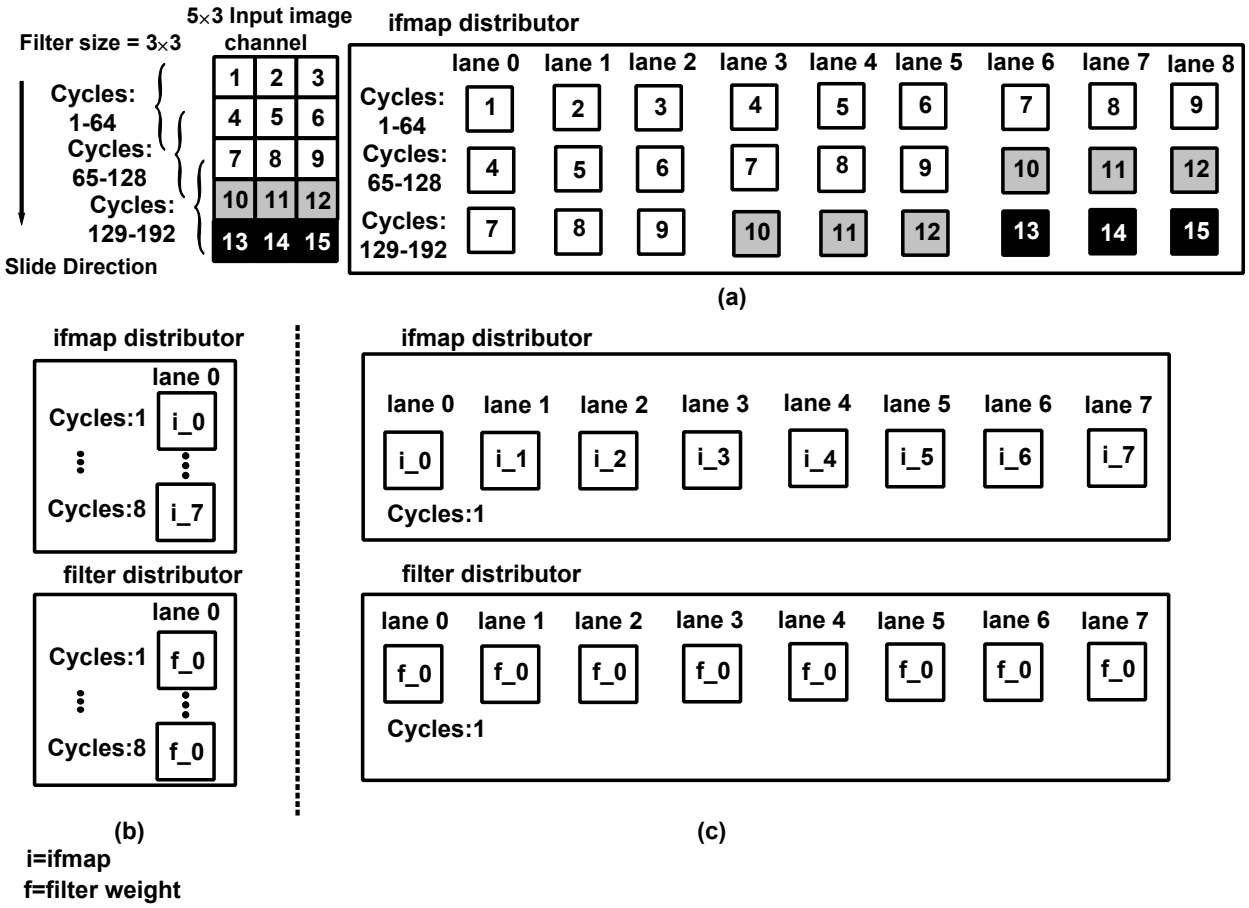


Figure 5.9: The proposed dataflow for (a) local reuse of the ifmaps in a PE while running convolutional layers, performing the computations of fully connected and depth-wise convolutional layers when (b) intra-level parallelism is not supported compared to (c) when intra-level parallelism is supported.

the internal control units of distributor units to feed the PEs with the newly loaded data from GBs or configures the distributor units to perform the computations of a new layer.

As explained in Section 5.5, the proposed architecture performs the computation of two pairs of filters at the same time for most of the layers which decrease the number of ifmap GB accesses. Regardless of performing the computations of 4 filters concurrently, our proposed dataflow is designed such a way to coordinate the computations of each specific layer type based on the feature of that layer type. For the regular convolutioal layers of benchmark networks, all the filter and ifmap rows with more than three elements are divided into rows containing three elements, and

zero padding is applied whenever needed. Regular convolutional layers come with the chance of local reuse of the ifmaps. For example, for a $3 \times 3$ filter and stride size one, the main control unit slides the filter vertically on each image channel from the top left corner to the down right corner and stores the partial sum in the partial sum GBs. Since the stride size is one, in the next processing cycle, the distributor unit only loads three new Ifmpas into the PEs to compute the next output feature. Figure 5.9 (a) shows the local ifmap reuse technique in our design while sliding a 3×3 kernel over a 5×3 input image channel to calculate three output features. In this example, the size of the stride is 1. To present the Figure 5.9 (a) more understandably, the Booth-encoded format of each ifmap that has 8 elements, is shown by a single number inside the input image channel. The signal lanes 0 to 8 in Figure 5.9 (a) are the outputs of ifmap distributor unit. These signals are considered for feeding the PEs located inside the first column of the computation engine with the loaded ifmaps. As explained in Section 5.2, the ifmap distributor unit feeds the PEs inside the computation engine with the Booth-encoded format of the ifmaps. Assuming that each ifmap and filter weight has 8 non-zero elements, it takes 64 clocks to calculate each of the output features in the proposed architecture. As shown in Figure 5.9 (a), the control unit inside the distributor unit shifts the contents of the mentioned signals and updates only the contents of 3 of the mentioned signals at cycles 65 and 129 with newly loaded ifmaps from the Global Buffer.

Our proposed dataflow also supports intra-value bit-level parallelism. This is possible since each PE produces the correct result even if the elements of a value are processed spatially instead of temporally. Figure 5.9 explains intra-level parallelism using an example. Figure 5.9 (b) shows the dataflow of the signal lane 0 both in the ifmap distributor unit and one of the filter distributor units of the computation engine proposed in [27] while performing the computations of fully connected layers and depth-wise convolutional layers. This signal in the ifmap distributor unit is considered for feeding the first lane of the PEs located in the first column of the computation engine in [27]. The signal lane 0 in the filter distributor unit is also considered for feeding the first lane of the PEs located in the first row of the computation engine in [27]. As shown in Figure 5.9 (b), the main control unit in [27] performs the computations of the elements of the Booth-encoded format of an ifmap and the first element of the Booth-encoded of a filter weight temporally. Although Figure 5.9 (b) shows the computations for just the first element of the filter weight, the dataflow for the rest of

the elements of the filter weight would be the same.

Figure 5.9 (c) shows the dataflow of the signal lanes 0 to 7 both in the ifmap distributor unit and one of the filter distributor units in our proposed design while performing the computations of fully connected layers and depth-wise convolutional layers. These signals in the ifmap distributor unit are considered for feeding the first 8 lanes of the PEs located in the first column of our proposed computation engine. The signal lanes 0 to 7 in the filter distributor unit are considered for feeding the first 8 lanes of the PEs located in the first row of our proposed computation engine. As shown in Figure 5.9 (c), the main control unit in our proposed design performs the spatial computations on both the Booth encoded format of an ifmap and the first element of the booth encoded filter weight. The same thing would be true for the rest of the elements of the Booth-encoded format of the filter weight. Accordingly, the results of these layers would be much faster in our design. As authors reported in [27], for the depth-wise convolutional layers and fully connected layers only 1/16 of the processing lanes is used which means the utilization of PEs is not efficient. However, as discussed above, our dataflow improves the speedup and resource utilization for such layers that come with a low chance of local reuse. The results in Section 5.7 show the achieved speedup of our architecture for fully connected layers of benchmark network over reference design is $\times 2.45$.

Furthermore, based on our proposed dataflow, the main control unit coordinates the computation of $1 \times 1$ convolutional layers by performing the computations of all the channels associated with each output feature in a depth-wise direction first and then slides the filter on the image channels vertically down from the top left corner to the down right corner of the input image channels. Accordingly, for $1 \times 1$ convolutional layers, the channels of the loaded ifmap and filters are divided into rows containing 4 elements, and zero padding is used wherever it was needed as well.

## 5.7 Results and Discussion

### 5.7.1 Implementation

In this chapter, we built a 16b version of the computation engine proposed in [27]. The reason we chose the accelerator in [27] as our reference design, is that this binary accelerator is very optimized in terms of performance and energy consumption compared to other binary accelerator

proposed until now [3, 30, 31]. The chosen reference design supports bit-wise and word-wise sparsity. In addition, the introduced PE in [27] is unlike other multiply-accumulate units very optimized in terms of area and power consumption. Like [27], a cycle-accurate simulator is developed based on our VHDL implementations to model both architectures. Furthermore, both designs are implemented in VHDL and are synthesized using SYNOPSYS Design Compiler and TSMC 65nm technology library. ARM Artisan 65nm SRAM and register file generator are used for creating the global buffer and scratchpads. The clock frequency is 200MHz and the core voltage is 1.1V for both designs. To measure the power cosumption precisely, the activity of the signals inside the computation engine are stored in a saif file during the functional simulation of the design. Then, the design is synthesized using Design Compiler and used the mentioned saif file to report the power of each unit and its sub-units.

### 5.7.2 Hardware Performance Evaluation

Five popular classifiers are used for evaluating the proposed architecture, namely VGG16 [15], GoogleNet [72], MobileNet V2 [60], SqueezeNet [7], and Xception [73]. The main reason for choosing these classifiers is to evaluate almost all different layer types of DNNs. The Batch size while doing inference for all the benchmark networks is one. The achieved speedups for different layers of benchmark networks, caused by equipping the reference design with the proposed technique in section 5.3 are shown in Figures 5.10 to 5.14. Due to the large number of layers of GoogleNet, MobileNet V2, SqueezeNet, and Xception only the achieved speedup for a few layers of these benchmark networks is shown in Figures 5.11 to 5.14. The average per layer speedup for all the benchmark networks is $\times 1.4$. In terms of different layer types, the average speedup while running regular and $1 \times 1$ convolutional layers is $\times 1.17$, and the average speedup while running depth-wise convolutional and fully connected layers is $\times 2.45$. In terms of different benchmark networks, the average per layer speedup while running different layers of VGG16, GoogleNet, and MobileNet V2 is $\times 1.33$, $\times 1.17$, $\times 1.71$, $\times 1.17$, and $\times 1.65$ respectively. Table 5.2 shows the consumed area by regular and shared PEs in our design. As mentioned, all the optimizations techniques used during the implementation of these PEs are the same and the only difference is the number of processing lanes inside these PEs.

Table 5.2: Comparison of area consumption of regular and shared PEs

| PE Specs. | Regular PE | Shared PE |
|---|---|---|
| Area ($\mu m^2$) | 10952 | 2577 |
| Processing lanes | 16 | 4 |

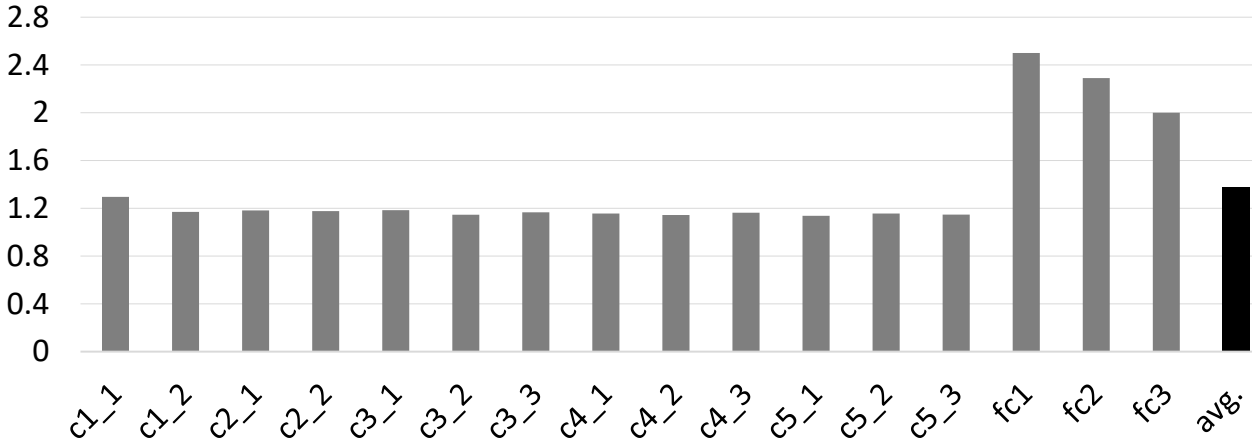


Figure 5.10: Speedup of layers of VGG16 by pruning the identical effectual computations.

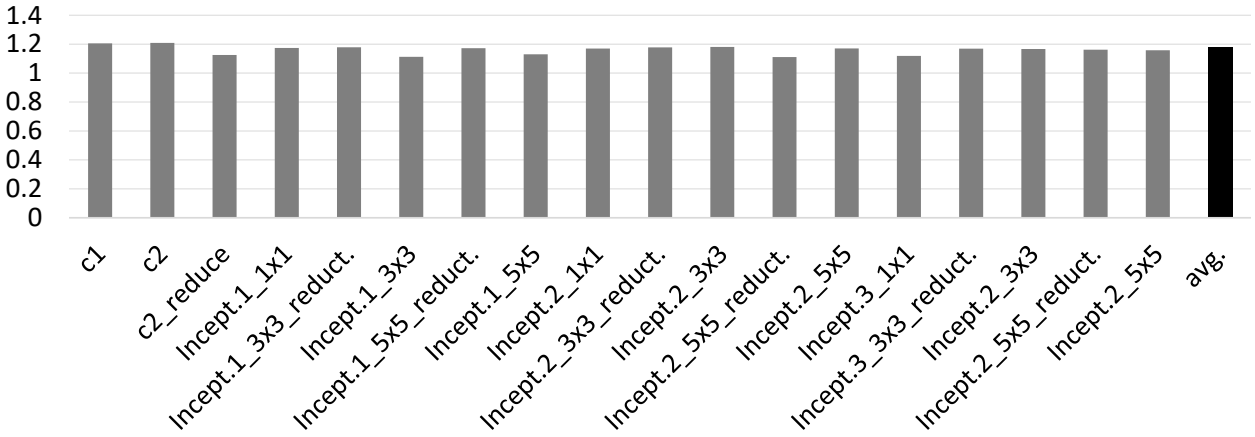


Figure 5.11: Speedup of a few layers of GoogleNet by pruning the identical effectual computations.

The throughput and energy consumption comparison of both designs is shown in Table 5.3. The average power consumption of the reference design and our proposed architecture is 43.65 mW and 43.92 mW, respectively. This simply means the power consumption of both designs is almost the same. In fact, although the number of MAC operations is decreased in our design but the power consumption of the whole design is still slightly increased due to the added logics.

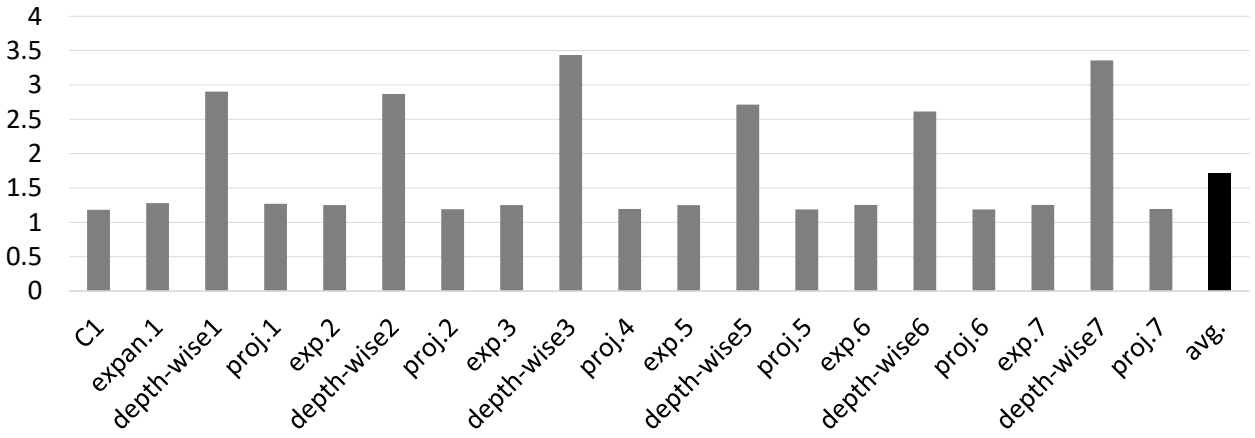


Figure 5.12: Speedup of a few layers of MobileNet by pruning the identical effectual computations.

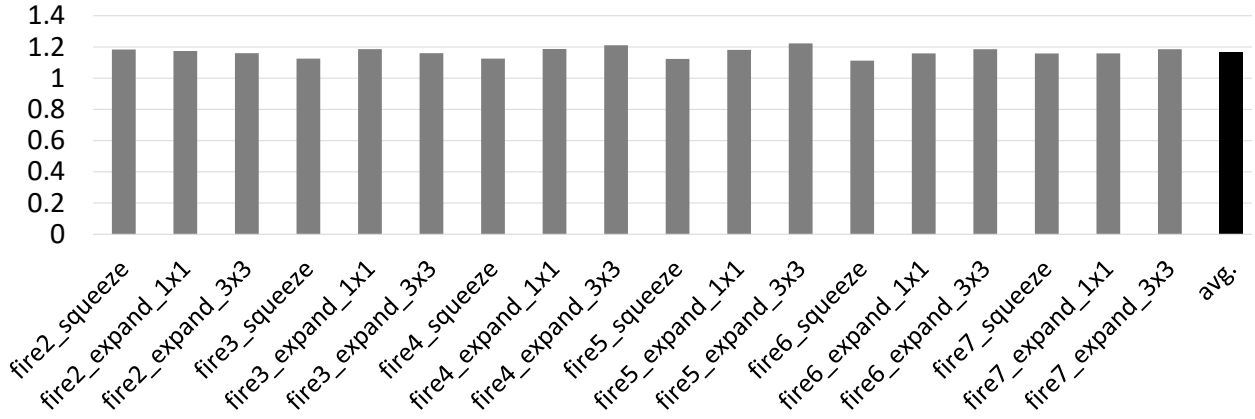


Figure 5.13: Speedup of a few layers of SqueezeNet by pruning the identical effectual computations.

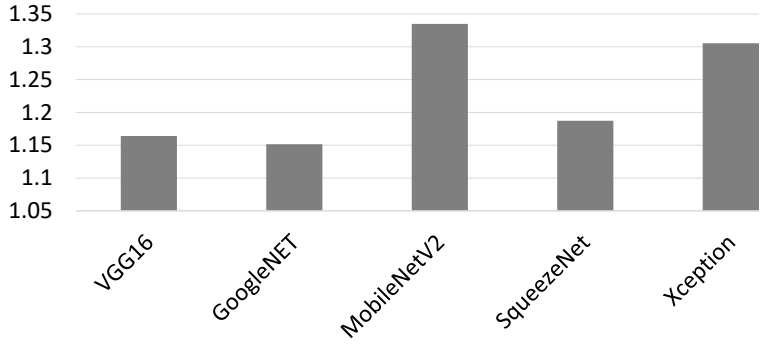


Figure 5.14: Speedup of a few layers of Xception by pruning the identical effectual computations.

However, our design come with an average per layer speedup of $\times 1.4$ which means less energy consumption while running benchmark networks. The PEs inside the computation engines account for 31.5% and 35.3% of the consumed power in the reference design and our proposed architecture

respectively. The average energy consumption of reference design and our proposed architecture for one inference while running VGG16 are 506 mJ and 434 mJ, while running GoogleNET are 57 mJ and 50 mJ, while running MobileNet V2 are 17 mJ and 13 mJ, while running SqueezeNet are 10.6 mJ and 8.9 mJ, and while running Xception are 186 mJ and 142 mJ, respectively. The energy efficiency of our proposed design while running benchmark networks is shown in Figure 5.15. The area overhead of applying our method compared to consumed area of the logic of the reference design is 24.75% and to the whole reference design 7.21%. The average Giga MAC operations (GMAC) per inference of the reference design and our proposed architecture for one inference while running VGG16 are 96 and 82.7, while running GoogleNET are 10.3 and 8.7, while running MobileNet V2 are 3.26 and 2.87, while running SqueezeNet are 2.3 and 1.98, and while running Xception are 34.3 and 29.7, respectively. The MAC operation in Table 5.3 refers to the element-wise MAC operation of the Booth-encoded of the ifmaps and filter weights. Furthermore, we believe composite metrics, such as performance per $\mu m^2$ and performance per watt provide a more detailed comparison between our proposed design and [27] in terms of efficient resource utilization and energy efficiency. The average performance per $\mu m^2$ of our design and the reference design [27] is 0.738 and 0.629 respectively and the average performance per watt of our design and the reference design [27] is 42.1 and 33.7 respectively. The comparison of these metrics are also summarized in Table 5.3.

We also compared our work to the state-of-the-art accelerators in terms of Giga MAC operation per inference [1, 28, 29]. The proposed architecture in Eyeriss [1] does not perform zero computations but also it does not skip those computations in time. As mentioned, our design not only skips bit-wise and word-wise sparsity but also skips identical MAC operations. The average Giga MAC operation per inference in our design is $\times$2.95 less than that of Eyeriss [1]. Also it is worth mentioning that, when our design is scaled to have the same area as that of Eyeriss [1], on average our design is $\times$3.5 faster and $\times$2.9 more energy-efficient than [1] while running VGG16. The proposed architecture in [28] exploits skipping sparsity by compressing the pruned weights, skipping computation cycles for zero-valued weights, but it still suffers from wasted computation cycles when the non-zero weight is to be multiplied with zero-valued activations. Furthermore, bit-wise sparsity is not explored in [28]. To give more detail, the average MAC operation per

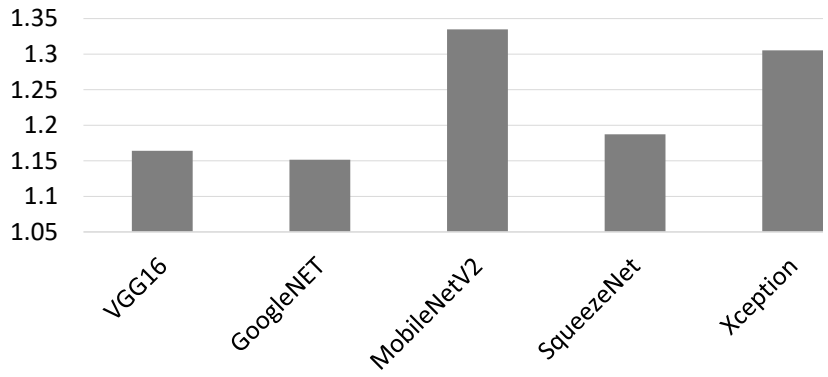


Figure 5.15: The average energy efficiency of the proposed architecture over the reference design while running benchmark networks.

inference in our design is ×9.8 less than that of [28] while running VGG16. The design frequency in [28] is 1 GHz. The maximum working frequency of our design is 900 MHz. When our design works with the highest frequency and is scaled to have the same area as [28], it achieves a speedup of ×3.1 and energy efficiency of ×2.8 while running conventional layers of VGG16. The proposed architecture in [29] skips computation cycles for zero-valued ifmaps, but it still suffers from wasted computation cycles when the non-zero ifmap is to be multiplied with zero-valued weights. When our design is scaled to have the same area as [29], the Giga MAC operation per inference and the power consumption of our proposed design is ×3.4 and ×1.25 less than that of [29] respectively. However, the proposed design in [29] is ×1.85 faster than our design. Although the authors in [29] did not evaluate their design while running depth-wise and fully connected layers that come with a low chance of data reuse, they believe their proposed dataflow is the main reason for their low computation latency while running DNNs. Furthermore, it is worth mentioning that it is possible to apply our proposed computation technique into the computation engine of these state-of-the-art accelerators to make them more energy-efficient.

Table 5.3: Comparison of throughput and energy consumption

| Hardware Specs. | | Reference Design [27] | | | | | Proposed Design | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Area (logic only) | | 243K ($\mu m^2$) | | | | | 303K ($\mu m^2$) | | | | |
| SRAM | Reg. file | 54kB | | | 1kB | | 54kB | | | 1.047kB | |
| Regular PEs | Shared PEs | 16 | | | NA | | 16 | | | 14 | |
| PRF Metric | | VGG16 | GoogleNet | MobileNetV2 | SqueezeNet | Xception | VGG16 | GoogleNet | MobileNetV2 | SqueezeNet | Xception |
| Infer./S | | 0.088 | 0.75 | 2.43 | 3.84 | 0.25 | 0.099 | 0.86 | 3.21 | 4.76 | 0.33 |
| Infer./J | | 1.97 | 17.34 | 56.9 | 94 | 5.36 | 2.3 | 19.97 | 75.96 | 111 | 7 |
| GMAC/Infer. | | 96 | 10.31 | 3.26 | 2.3 | 34.3 | 82.7 | 8.7 | 2.83 | 1.98 | 29.7 |
| Avg. PRF/utilized area | | 0.629 | | | | | 0.738 | | | | |
| Avg. PRF/consumed power | | 33.7 | | | | | 42.1 | | | | |

# 6. Efficient Approximate DNN Accelerators for Edge Device

Equipping edge devices with Deep Neural Networks (DNNs) could result in a revolution in human interactions with surrounding environments as edge devices would be able to perform more complex tasks. However, DNNs are power-hungry, performing billions of computations in terms of one inference. Applying approximate computing techniques reduces the cost of the underlying circuits so that DNN inferences would be performed more efficiently where negligible inference accuracy loss is acceptable. There are many approximate multipliers proposed for various applications until now. However, the function of only a few of these approximate designs has been explored while performing inference of DNNs. Furthermore, little attention has been given on applying various approximation techniques into different layers of DNNs.

In what follows, Section 6.1 proposes a re-configurable MAC unit suitable for inference of DNNs. To add more detail, Subsection 6.1.2 provides some background information on radix-4 Booth multiplication. Then it presents the considered step-wise approach for designing a re-configurable approximate Booth multiplier. Subsection 6.1.3 explains the proposed Booth multiplier architecture and the general architecture of the accelerator is built upon the explored approximate multipliers in this chapter. Subsection 6.1.4 compares the performance of the developed

---

A major portion of the first section of this chapter is going to be appeared in M. Asadikouhanjani, H. Zhang, and S.-B. Ko, "Efficient Approximate DNN Accelerators for Edge Devices: An Experimental Study," Submitted as a book chapter to Springer (to be appear by August 30th, 2022).

M.A conceived of and designed the proposed accelerator. M.A carried out all the simulations, implementations, and analysis. M.A wrote the manuscript in consultation with S.K and H.Z.

Booth multiplier to the most competitive designs in terms of area and energy consumption.

Furthermore, in Section 6.2 the tolerance of different layers of DNNs to various degrees of approximation is evaluated. To provide more information about the content of this section, Subsection 6.2.1 provides a brief introduction, the motivations behind the content of this section are explained in Subsection 6.2.2, Subsection 6.2.3 proposes a re-configurable architecture based on FPGA platforms, and lastly Subsection 6.2.4 evaluates the effect of applying various degrees of approximation at run-time on inference accuracy of DNNs.

## 6.1 Re-configurable Approximate MAC Unit

### 6.1.1 Introduction

Although the training phase of DNNs comes with billions of Multiply and Accumulate (MAC) operations when compared to inference. But, it is the inference that is usually subject to more strict design constraints. For example, equipping edge devices with DNN applications that usually come with resources and energy constraints. Accordingly, efficient implementation of DNNs is vital before integrating them into edge devices [74].

Many proposed DNN accelerators until now have shown significant improvements while performing the computations of DNN inferences [3, 17]. In [54], authors have shown that performing the computations in memory results in remarkable energy efficiency. These types of accelerators try to avoid data movement between memory and processing engine. However, their non-generic layouts are expensive to build since it is usually a mix of CMOS and other technologies. The authors in [23] and Eyeriss-V2 [24] have shown that efficient Processing Element (PE) and Network on Chip (NoC) designs are both critically important to keep the performance of the whole design high in the NoC-based DNN accelerators. There are also other works that have focused on efficient data movement inside the computation engine [75].

Authors in [69, 76] have proposed an algorithm to reduce the number of multiplication operations. Eyeriss [1] is an efficient accelerator equipped with the row stationary dataflow which increases the chance of local reuse of data inside the computation engine.

In [27], the authors have presented a method for transparently identifying ineffectual computations during inference with deep learning models. Specifically, by decomposing multiplications down to the bit level. The proposed architecture in [27] targets bit sparsity, that is zero bits since processing zero bits in a MAC operation does not affect the outcome. However, in these types of accelerators, the focus is only on skipping the ineffectual computations [30, 31]. In [77], the authors have proposed a real-time architecture for skipping non-effectual and possible effectual computations. It has been shown that when the MAC operations are decomposed down to bit level, there is still room to prune identical effectual computations.

One of the promising solutions to perform inference of DNNs efficiently is utilizing approximate multipliers inside the computation engine of DNN accelerators [32]. Previous authors have reported that DNNs are resilient against small arithmetic errors [33]. Authors in [51] have shown that the costs of applying DNNs to edge devices can be mitigated via approximate computing. Exploiting the resilience of DNNs to numerical errors from approximate computing, they have demonstrated a reduction in communication overhead of distributed deep learning training via Adaptive residual gradient Compression (AdaComp), and reduction in computation cost for deep learning inference via Parameterized clipping Activation (PAct) based network quantization.

In [78], authors have proposed a novel automated framework for High-Level Synthesizing (HLS) of approximate accelerators by providing a library of approximate functional units. Authors in [79] have proposed a novel approximate multiplier that can be used for reducing the energy consumption of MAC-oriented signal processing algorithms. To give more detail, two approximate 4 to 2 compressors with opposite error directions, have been developed to minimize the hardware costs, and then the corresponding approximate multipliers have been analyzed to predict the amount of error in a probabilistic way. Compared to previous works, suffering from the accumulated errors, the proposed interleaving method in [79] generates a narrow and balanced error distribution.

In [34] authors have introduced approximate multipliers based on alphabet-set multiplication. The weights are divided into parts, having 4 bits. Multiplication by each 4-bit part of the weight is implemented by shifting a pre-computed input value and followed by summation. This architecture uses an algorithm to approximate multiplication in the specified PEs of the computation engine when training DNNs. Similarly in [50], authors have proposed an algorithm that determines bit

precision for all PEs to minimize power consumption for given target accuracy. While all of the proposed algorithms in [34, 50] are very progressive and interesting research topics that propose different paradigms of efficient DNN processing, it is not easy to integrate them into the edge devices equipped with DNN applications.

In [52], the authors have evaluated how the error caused by an approximate logarithm-based multiplier affects deep DNN inferences. In more detail, the authors have explained that how convolution and fully connected layers maintain their intended functionalities despite applying a logarithm-based approximation into the computation engine of DNN accelerators. Also, they have explained that the total computed error will not converge properly while performing the MAC operations of different layers using approximate adders. In this chapter, a detailed experimental step-wise approach for designing a re-configurable approximate Booth multiplier suitable for performing inference of DNNs. Furthermore, applying various approximation techniques into different layers of DNNs is investigated.

### 6.1.2 Designing a Re-Configurable Approximate Booth Multiplier for Inference of DNNs

DNNs can work with smaller data types with less precision, such as 8-bit integers, which makes them suitable for integration into edge devices [80]. In other words, it is possible to achieve an acceptable accuracy by utilizing 8-bit integer multipliers instead of regular 32-bit floating-point multipliers. Radix-4 Booth multiplication is a popular multiplication algorithm that reduces the size of the partial product array by half. Figure 6.1 (a) shows the dot-diagram of a signed 8×8 radix-4 Booth multiplier. As the considered bit-width of the multiplier operand is 8, the radix-4 Booth encoded of each weight will have 4 elements. In Figure 6.1 (a), the partial product bits are organized into 4 rows and 16 columns. In this figure, the numbers given to the rows and columns of partial products are based on their significance.

The weights of each layer of DNNs have a Gaussian distribution around zero which makes them suitable to be encoded as the multiplier operand in the radix-4 Booth multiplication algorithm. To add more detail, by choosing weights as the multiplier operands, the produced error caused by applying different approximation techniques into partial products of the multiplier has opportunities

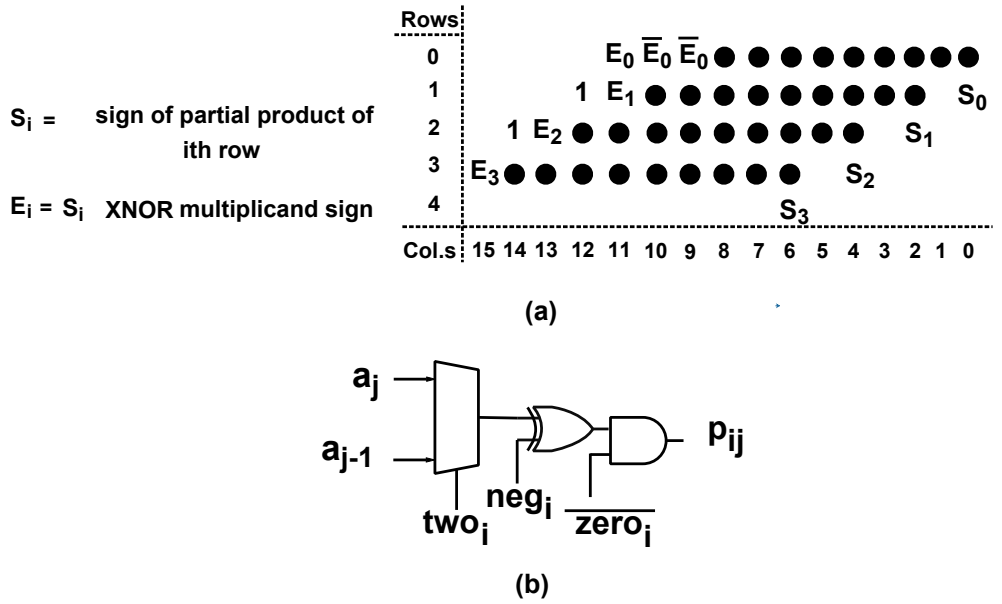


Figure 6.1: The (a) dot-diagram of a signed 8×8 radix-4 Booth multiplier and the (b) internal architecture of the partial product generator in a radix-4 Booth multiplier. ● : exact partial products.

to be both positive and negative which accordingly causes an unbiased distribution of error around zero. Unbiased distribution of error has a direct effect on inference accuracy, especially, large DNNs. Authors in [81, 82] provided a detailed comparison.

Figure 6.1 (b) shows the partial product generator circuit in radix-4 Booth algorithm which tends to be more logically complex than a single AND-gate [9]. However, the Booth multiplication algorithm makes up for this in that it reduces the actual number of partial products that need to be generated and therefore accumulated [9]. In this Figure 6.1 (b), $\{a_{i-1}, a_i\}$ are two following bits of each input feature map (ifmap) and $\{p_{ij}\}$ is $j^{\text{th}}$ bit of the $i^{\text{th}}$ row of partial products. As mentioned earlier in this section, the multiplier operand is grouped in sets of bits $\{b_{2i-1}, b_{2i}, b_{2i+1}\}$ which corresponds to one of the values from 0, ±1, ±2. The radix-4 Booth encoder encodes these bit groupings into three signals $neg_i$, $two_i$, and $zero_i$. Table 6.1 illustrates how the encoding terms and multiplier bits are associated with radix-4 Booth terms.

To evaluate the function of the multipliers explored in the following subsections, an accelerator is been implemented which is discussed in detail in Subsection 6.1.3. This accelerator can perform

Table 6.1: Radix-4 Booth terms corresponding to various inputs and encoded signal values.

| Inputs | | | Encoded Signals | | | Value |
|---|---|---|---|---|---|---|
| $b_{2i+1}$ | $b_{2i}$ | $b_{2i-1}$ | $neg_i$ | $two_i$ | $zero_i$ | |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | +1 |
| 0 | 1 | 0 | 0 | 0 | 0 | +1 |
| 0 | 1 | 1 | 0 | 1 | 0 | +2 |
| 1 | 0 | 0 | 1 | 1 | 0 | −2 |
| 1 | 0 | 1 | 1 | 0 | 0 | −1 |
| 1 | 1 | 0 | 1 | 0 | 0 | −1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 |

the computations of different layers of DNNs. The output of this accelerator is an input for a Softmax calculator. As an example, a picture of a Crane which is shown in Figure 6.3 (a), is given to this accelerator when it runs Squeezenet [7]. SqueezeNet is an object image classifier that classifies 1000 object images. Crane is class number 135 of the SqueezeNet. Figure 6.2 (a) shows the computed output of the proposed accelerator when exact Booth multipliers are used. In this figure, index 135 which represents the Crane class, has the highest value among other indexes. In addition, the value of most of the non-relevant classes is pretty low compared to the peak index.

There are many different types of approximate multipliers and MAC units out there, with various costs and error characteristics [9, 83–87]. Without studying the error characteristics of an approximation method, there is a low chance to get the optimum accuracy after applying that specific approximation into DNN accelerators. Furthermore, it is necessary to study the error characteristics of approximate design using real benchmark DNNs. Since applying approximation into the computation engine of DNN accelerators is somehow different. In more detail, DNNs come with a high number of layers that each layer has various number of channels. Also, the error could propagate across layers. In what follows, the effect of the applied some common approximate techniques is discussed in more detail by calculating and analyzing the error distribution, Normalized Mean Error Distance (NMED), and variance of the error.

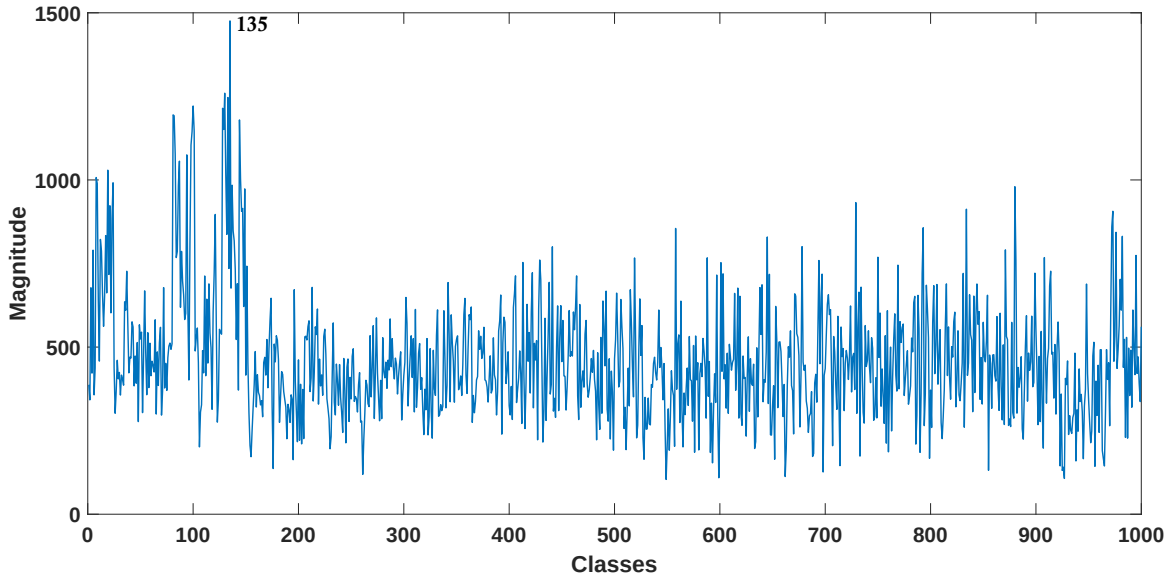


Figure 6.2: Output of the proposed accelerator in Subsection 6.1.3 when a picture of a Crane is fed into this accelerator that runs Squeezenet in exact mode.


Figure 6.3: Pictures of a (a) Crane, a (b) Black Swan, and a (c) Feather Boa.

To analyze the error characteristics of different approximation methods, various approximate multipliers are simulated based on the convolution operation using $f_w \times f_h$ filter kernels, where the internal MAC operations can be formulated as $\sum_{l=1}^{c} \sum_{i=1}^{w} \sum_{j=1}^{h} I_{lij} \times K_{lij}$. Note that the number of accumulated multiplication results varies with the number of channels in the convolution operations, denoted as C. Similar to authors in [79] for each filter, the accumulated error would be as (1). In

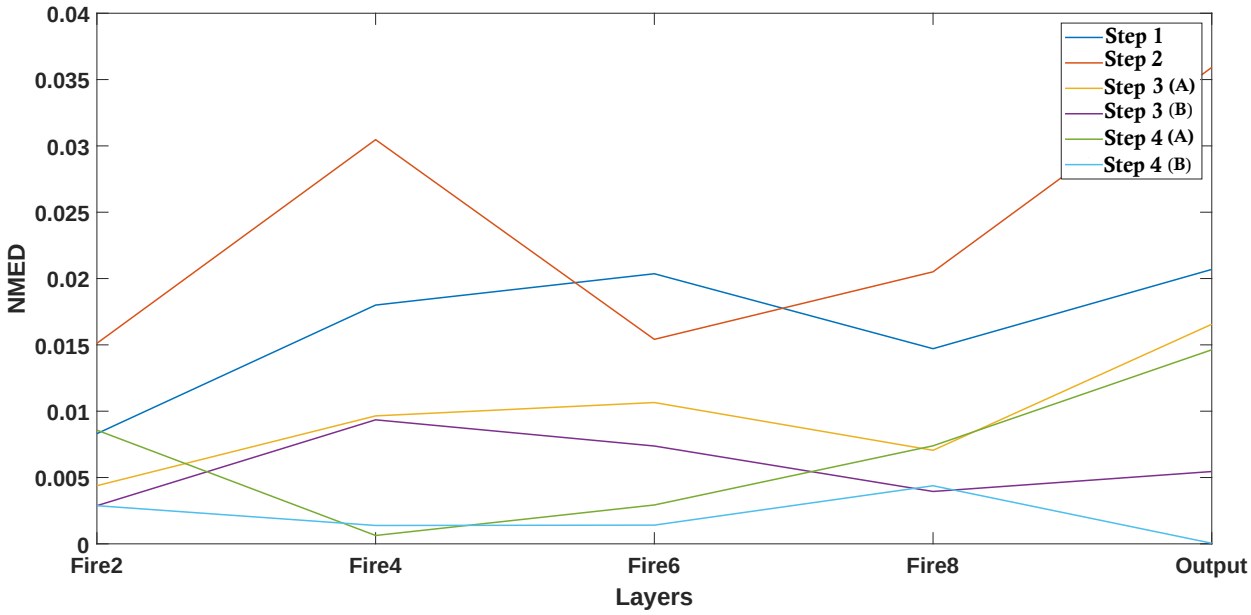


Figure 6.4: The computed NMED of a few layers of SqueezeNet [7] for the variuos approximation techniques explored in Subsection 6.1.2 while running ImageNet dataset [8].

addition, the equations shown in (2) are used to calculate the NMED.

$$E_{MAC} = \sum_{l=1}^{c} \sum_{i=1}^{w} \sum_{j=1}^{h} Approx.(I_{lij} \times K_{lij}) - (I_{lij} \times K_{lij}) \tag{6.1}$$

$$NMED = \frac{Mean(|E_{MAC}|)}{MAX(\sum_{l=1}^{c} \sum_{i=1}^{w} \sum_{j=1}^{h} I_{lij} \times K_{lij})} \tag{6.2}$$

**Step 1, Removing Correction Terms**

Each signal $S_i$ shown in Figure 6.1 (a) is a correction term that is identical to signal $neg_i$ shown in Figure 6.1 (b). Removing these correction terms from partial product rows comes with decreasing the length of the critical path by $0.18ns$ inside the design shown in Figure 6.1 (a). However, it also comes with increasing the magnitude of each negative partial product row slightly. It is like, performing 1's complement instead of 2's complement. Figure 6.5 is the output of the proposed accelerator in Subsection 6.1.3 which shows the computed value of all the object classes when this accelerator is fed by a Crane picture shown in Figure 6.3 (a), and all the correction terms

are removed from the Booth multipliers. In this figure, index 101 is the peak index. This index represents the Black Swan class which is shown in Figure 6.3 (b). Index 135 is still among the 5 indexes with the highest values though.

Figure 6.4 shows the NMED of 4 hidden layers and the final Average-pooling layer of SueezeNet for different explored approximation techniques in this section. The input test-benches are created using ImageNet validation data-set [8]. To present all the values in Figure 6.5 in a more readable manner, the NMED values of the output layers for all the approximate methods in this figure, are divided by 32. As expected, when all the correction terms are removed, the normalized error distance is high, especially for the layers with a high number of channels.

As this approximation only increases the magnitude of negative partial products, it seems that the error caused by applying this method should only be distributed on negative values. However, this is only true for the first layer of DNNs. Figure 6.6 (a) shows the error distribution of layer Fire 4 of Squeezenet when all the correction terms are removed and the DNN is fed by the mentioned Crane picture above. Based on this figure, the error is mostly distributed on negative values, however, the error is still distributed on positive values too. This is caused by the propagated error from former layers. To add more detail, since the number of output features that are set to zero after applying non-linearity increases in former layers, the input image of hidden layers would be different compared to the exact mode. The variance of error distribution for this layer is $1.13e+03$.

This technique can be used in the layers where its effect is negligible on DNN inference accuracy such as layers that contain a few number of channels, or when the Booth-encoded element of the weights are mostly positive, especially those regarding the least significant partial product rows. Since in such layers, the sum of the accumulated error would not be that large enough to affect the inference accuracy. However, as shown in the following steps, that it is possible to apply this technique into large DNNs if the effect of this approximation is somehow will be compensated by other applied approximation techniques.

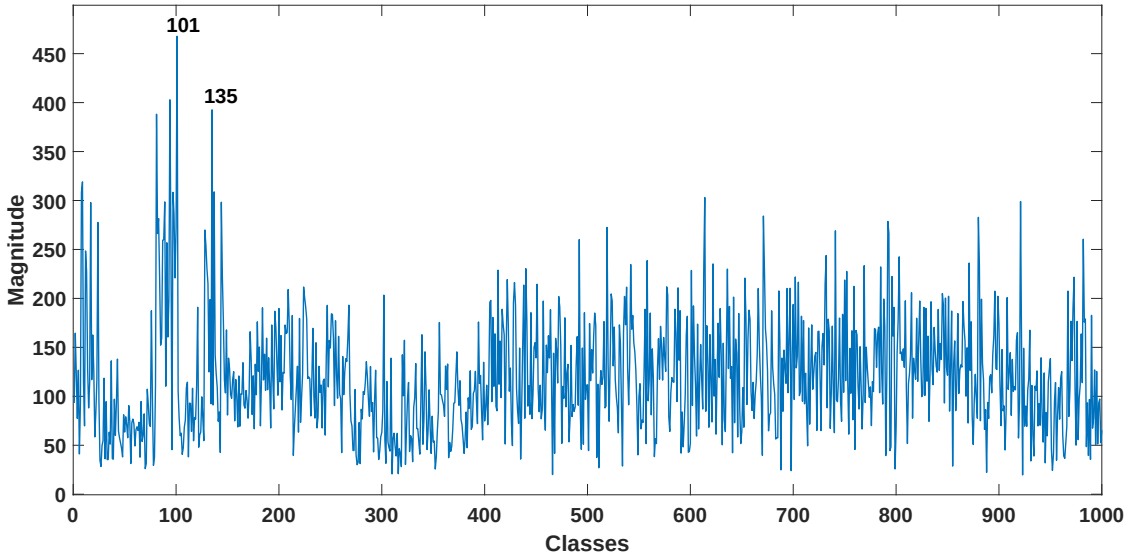


Figure 6.5: Output of the proposed accelerator when a picture of a Crane is fed into this accelerator, and all the correction terms are removed in all the Booth multipliers.

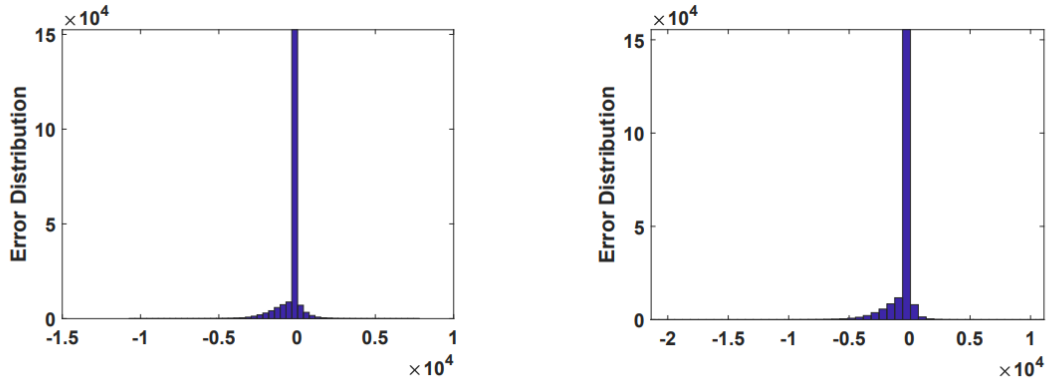


Figure 6.6: The error distribution of Layer Fire 4 when (a) all the correction terms are removed, and when (b) the 5 least significant partial product columns are truncated in all the Booth multipliers.

### Step 2, Truncation of Partial Products

Truncation is one of the most efficient ways of applying approximation into multipliers in terms of reducing power consumption, area, and complexity. Replacing a few partial products in the least significant columns of the multiplier with zeros comes with removing the circuit that generates and the circuit that is considered for the accumulation of those partial products. Furthermore, truncation of the least significant columns in a Booth multiplier also means reducing the magnitude

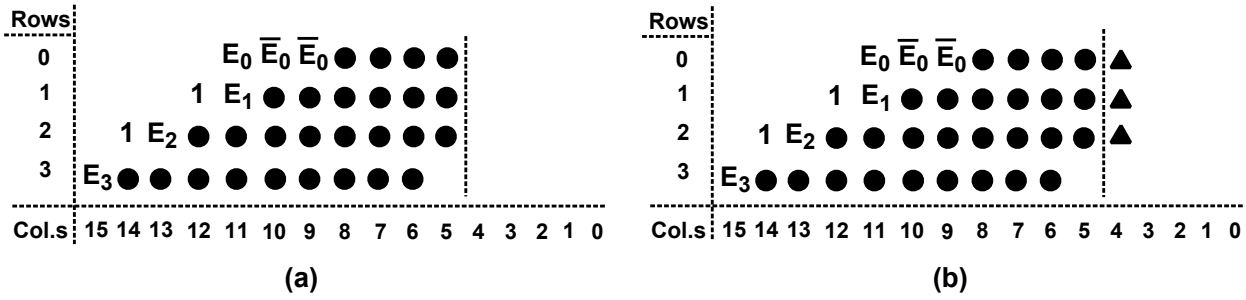


Figure 6.7: The dot-diagram of an approximate signed radix-4 Booth multiplier explored in step 1, when (a) the 5 least significant partial product columns are truncated, and (b) the partial products in the 5 least significant columns are replaced with a single inexact partial product [9]. ▲ : inexact compressed partial products using an $OR$ reduction operation. ● : exact partial products.

of positive partial products and increasing the magnitude of negative partial products. Therefore, the error direction for many of the output features, especially those in the high-depth layers would be negative. Combining this approximation with the approximation explored in the previous step, the total accumulated error direction would still be toward negative.

Figure 6.7 (a) shows the dot-diagram of a signed 8×8 radix-4 Booth multiplier when 5 least significant columns of partial products are truncated in excess of the introduced approximation technique explored in the previous step. Figure 6.8 shows the output of layer Fire 4 when the picture of a crane shown in Figure 6.3 is fed into the proposed accelerator, and the introduced approximate techniques both in steps 1 and 2, are applied into all the Booth multipliers of our proposed accelerator. In Figure 6.8, not only there is no obvious peak at index 135, but also this output is noisy. In more detail, the computed value of so many object classes that are not even close to the given input image is much higher than that of the previous design. In this figure, the peak index is 553, this index represents the Feather Boa class which is shown in Figure 6.3 (c).

As expected in Figure 6.4, NMED increases for so many layers, especially those with a large number of channels. Figure 6.6 (b) shows the distribution of the error for layer Fire 4 of Squeezenet when both approximation techniques in the previous steps are applied into the 8×8 radix-4 Booth multipliers, and the Crane picture shown in Figure 6.3 (a) is fed to the DNN accelerator as an input. Similar to the previous step, the error is mostly distributed on negative values. The variance of this distribution is $1.43e+03$, which is even more than that of the first step. Although, the error

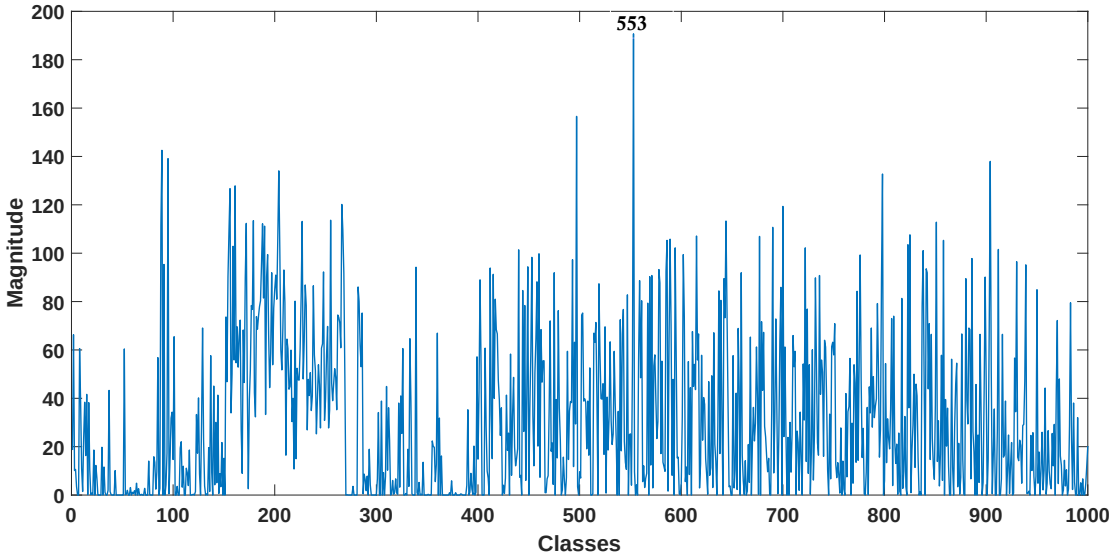


Figure 6.8: Output of the proposed accelerator when a picture of a Crane is fed into this accelerator, and the introduced approximation in steps 1 and 2 are applied into the Booth multipliers.

direction of the explored approximation techniques in steps 1 and 2, is toward negative values, it is still possible to use these approximation techniques in the layers that their number of channels is low. Furthermore, as explained in the following steps, it is applicable to use these types pf approximations with other approximate techniques that come with a positive error direction, since there is a chance that the total accumulated error would be balanced around zero.

**Step 3, Replacing Exact Partial Products with Inexact Ones**

**Step 3 (A)**

By decreasing the number of truncated partial product columns, the inference accuracy increases for the explored design in the previous step. However, this step shows that by studying and considering the behavior of the total accumulated error, there is a high chance to end up to a design which is not only very efficient in terms of area and power consumption but at the same time accurate. Authors in [9], have proposed an approximation technique for the partial product generation stage. However, they have not evaluated their method by running any DNN as their benchmark. In more detail, a group of exact bits of the multiplicands replaced with a single inexact partial product using an $OR$ reduction operation.

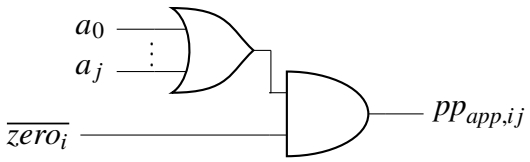


Figure 6.9: The gate-level architecture of the approximate partial product generator which compresses a few partial products into a single inexact partial product using an $OR$ reduction operation [9].

Figure 6.9 shows the gate level architecture of this approximation technique. The partial product $pp_{ij}$ is computed by performing an $OR$ reduction on the $j+1$ least significant bits of multiplicand and finally checking the $zero_i$ signal in case the partial product ought to be zeroed out. Figure 6.7 (b) shows the dot-diagram of a 8×8 radix-4 Booth multiplier when, in each of the least 3 significant rows, the partial product bits that are in the 5 least significant columns of the dot-diagram are replaced with a single inexact partial product. As this figure shows, this approximation is applied in excess of the applied approximation in step 1.

Applying the approximation technique in [9], comes with reducing the magnitude of positive partial product rows that the logical value of their $j^{\text{th}}$ bit and also the logical value of at least one of the bits $0, ..., (j-1)$ is "1". However, this approximation technique increases the magnitude of many positive partial product rows that the logical value of their $j^{\text{th}}$ bit is "0" and at least the logical value of one of the bits $0, ..., (j-1)^{\text{th}}$ is "1". In addition, this approximation technique would increase or decrease the magnitude of negative partial product rows in the same way as well. Furthermore, as shown in Figure 6.9, the multiplicand operand is approximated before applying $neg_i$ signals, and since the weights that are selected as the multiplier operands in the multiplication operations, have a Gaussian distribution around zero, the error direction would be balanced around zero as well. Moreover, although the error direction caused by the discussed approximate techniques in the previous step, is toward negative values for many of the output features. But, combining the explored approximations in both steps, the total accumulated error distribution is more balanced around zero compared to just applying the approximation methods explored in the previous steps.

Figure 6.11 shows the output of the proposed accelerator when the same Crane picture used for bench-marking former approximate architectures, is used as an input of the accelerator, and the

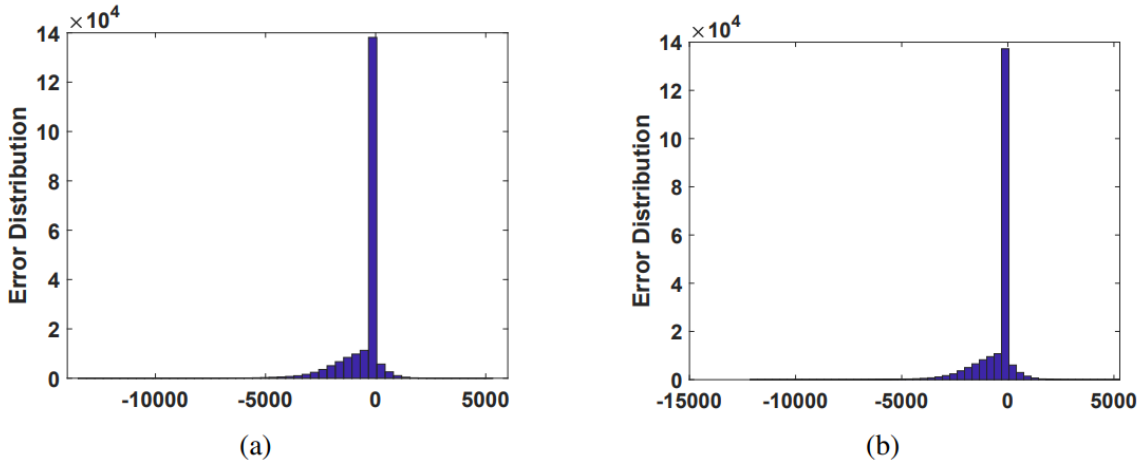


Figure 6.10: The error distribution of Layer Fire 4 when (a) the partial products in the 5 least columns are compressed to a single inexact partial product using an $OR$ reduction operation, and when (b) the 5 least significant partial product are replaced with a constant logical value "1" in all the Booth multipliers. The mentioned approximation is applied in excess of the explored approximation in step 1.

approximation technique shown in Figure 6.7 (b) is applied to all the multipliers of this accelerator. In this figure, there is an obvious peak at index 135 and the value of other object classes is pretty low than that of the explored design in the previous step. Figure 6.10 (a) shows the error distribution of layer Fire 4 of the Squeezenet while a group of exact bits in the 5 least significant columns of the dot-diagram replaced with a single partial product in excess of applied approximation techniques in step 1. The variance of this distribution is $0.99e+03$. This distribution is more balanced compared to previous methods but not significantly. However, for some other layers like Fire 2, the error distribution is more balanced around zero compared to explored approximation in the previous steps.

Figure 6.4 shows the NMED of a few layers of Squeezenet when the approximation techniques shown in Figure 6.7 (b), is applied to the multipliers of our proposed accelerator. As expected, by replacing a group of exact bits of partial product rows with a single inexact partial product and removing all the correction terms, the computed NMED of most of the layers is smaller compared to only applying the discussed approximation techniques in the previous steps. Furthermore, the propagated error from low-depth layers to high-depth layers is less as well.

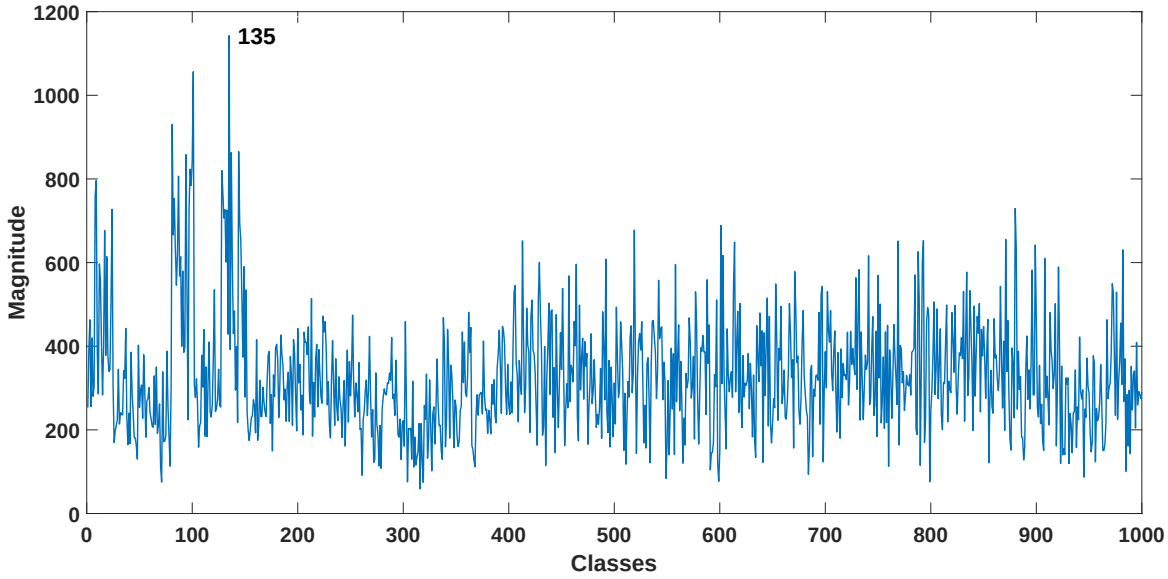


Figure 6.11: The output of the proposed accelerator when a picture of a Crane is fed into this accelerator, and the partial products in the 5 least columns are compressed to a single inexact partial product using an $OR$ reduction operation in all the Booth multipliers. The mentioned approximation is applied in excess of the explored approximation in step 1.

**Step 3 (B)**

It is possible to implement a more efficient design by rethinking the mentioned approximate technique shown in Figure 6.7 (b). This approximation technique increases the magnitude of so many positive and negative partial product rows. However, some partial product rows still remain unchanged. As an example, the magnitude of the partial product rows that their 5 least significant columns are zero, is identical before and after applying approximation. Based on our evaluation, the magnitude of unchanged partial products must change as well to have better accuracy. Replacing the inexact partial product in the first and second row of the approximate dot-diagram shown in Figure 6.7 (b) with a constant logical value "1", increases the magnitude of unchanged partial products as well. This change is also very efficient in terms of implementation. Since, for all the multipliers, the $OR$ reduction operation is replaced by logical constant "1". Figure 6.13 (a) shows the dot-diagram of this approximation technique.

Figure 6.12 shows the output of the proposed accelerator when the same Crane picture that is used for bench-marking the former approximate multipliers, is fed to this accelerator, and all its

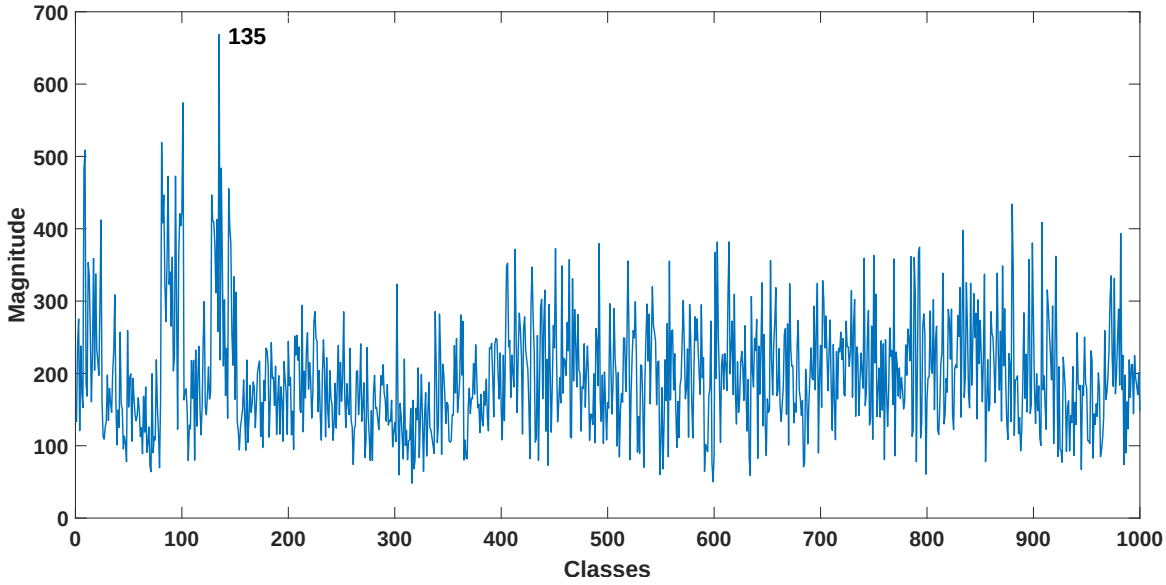


Figure 6.12: The output of our proposed accelerator when a picture of a Crane is given to this accelerator, and the partial products in the 5 least columns are replaced with a single constant logical value "1". The mentioned approximation is applied in excess of the explored approximation in step 1.

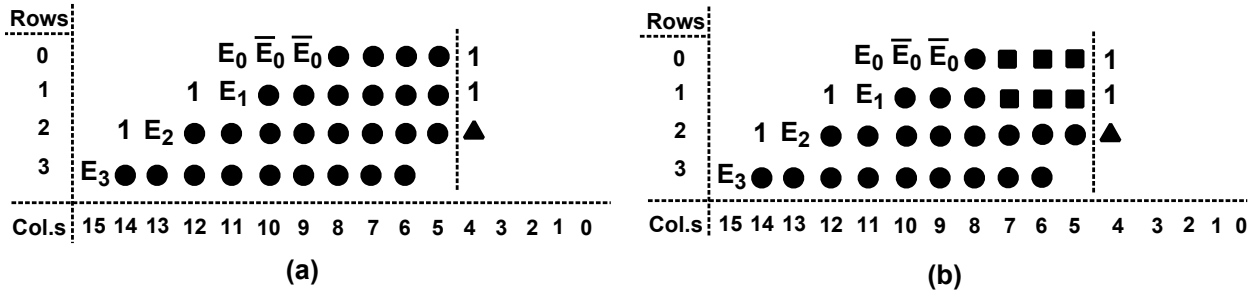


Figure 6.13: The dot-diagram of an approximate signed radix-4 Booth multiplier explored in step 1, when (a) the 5 least significant partial product columns are replaced with a constant logical value "1", and when (b) approximate partial product generators are used in the columns 5 to 7 of the 2 least significant rows. ▲ : inexact compressed partial products using an $OR$ reduction operation. ■ : inexact partial product generator by removing the multiplexers considered for shifting two adjacent partial product bits. ● : exact partial products.

multipliers are equipped with the approximation technique shown in Figure 6.13 (a). As expected, when the multipliers of the proposed accelerator are equipped with the approximate technique

shown in Figure 6.13 (a) and Figure 6.7 (b), the computed probability using the Softmax function provided by Matlab for index 135, is 94% and 85% respectively.

Figure 6.10 (b) shows the error distribution of layer Fire 4 of the Squeezenet when the inexact partial product in the first and second row are replaced with logic value "1". The variance of this distribution is $0.94e+03$ which is lower than that of the previous step. Similar to the proposed method in [9], the error distribution shown Figure 6.10 (b) is more balanced compared to previous steps, but not significantly. However, for many other layers such as Fire 8, the error distribution is much more balanced around zero compared to explored approximation in the previous steps.

Based on Figure 6.4, when exact partial products in the first and the second row are replaced with constant logical value "1", the computed NMED of the hidden layers of Squeezenet decreases. This figure also shows that the accumulated error that is propagated from low-depth layers to high-depth layers, is lower compared previous explored designs, especially for layers with high number of channels.

**Step 4, Applying Dynamic Approximation into Partial Products While Performing Shift Operation**

**Step 4 (A)**

In this step, the design which its dot-diagram has shown in Figure 6.13 (a) is pushed further. The authors in [9], have explored the effect of removing the multiplexer in the exact partial product generator shown in Figure 6.1 (b). However, they have not evaluated their method by running any DNN as their benchmark. In more detail, applying this approximation into j least significant partial products of the $i^{\text{th}}$ row means if the value of the $i^{\text{th}}$ Booth encoded element of the multiplier operand is +2 or −2, the shift operation will not be applied on the j least significant partial products of the $i^{\text{th}}$ row of the Booth multiplier.

Similar to the former step, this approximation is applied before applying the $neg_i$ signals, meaning the error distribution caused by this method, depending on the weights, has the potential to have a balanced distribution around zero. However, this method is somehow different since the value of only a few percentages of the whole Booth-encoded weights of the DNNs are +2 or

−2. This issue has not been considered in [9]. For example, the value of only 2% of the fourth Booth-encoded element of the weights of the Squeezenet is +2 or −2. Accordingly, applying this approximation into the fourth row of the partial products only affects a low percentage of the output features in Squeezenet. However, the fourth partial product row is the most significant row of the design which its dot-diagram shown in Figure 6.13 (a). Based on our evaluation, applying this approximation to the fourth row of the partial product of the Booth multipliers leads to a sharp inference accuracy loss while running SqueezeNet on the proposed accelerator. A high percentage of the Booth-encoded elements of the weights that their value is +2 or −2, is among the 2 least significant elements of the weights of the benchmark networks.

By applying the approximation technique explained above, depending on the logical value of $j-1^{\text{th}}$ bit, the magnitude of the partial product rows could increase or decrease after a shift operation. For example, if the $i^{\text{th}}$ partial product row is positive and the logical value of $j-1^{\text{th}}$ bit is "1", the magnitude of the partial product will increase after applying a shift operation. Since the logical value of $j-1^{\text{th}}$ bit will be repeated as $j^{\text{th}}$ and $j-1^{\text{th}}$ bits of that partial product row through a shift operation. The same thing is true when the partial product row is negative and the logical value of $j-1^{\text{th}}$ bit is "0". When the logical value of $j-1^{\text{th}}$ bit is "0" and the partial product is positive, the magnitude of the partial product will decrease after applying shift operation. The same thing is true when the partial product is negative and the logical value of the $j-1^{\text{th}}$ bit is "1".

Figure 6.13 (b) shows the dot-diagram of the Booth multipliers when the multiplexers inside columns 5 to 7 of the 2 least significant rows are removed in excess of the approximation techniques evaluated in the previous step. Figure 6.14 (a) shows the error distribution of layer Fire 4 of the Squeezenet when the approximation technique shown in Figure 6.13 (b), is applied to all the multipliers, and the Crane picture shown in Figure 6.3 (a) is fed to the proposed accelerator. This distribution is more balanced around zero compared to previous methods. But the variance of this distribution is $1e+03$ which is slightly larger than that of the designs explored in the previous step. However, for the high-depth layers with a large number of channels, the error distribution is not balanced around zero.

Figure 6.4 (a) shows the NMED for a few layers of Squeezenet when the approximation techniques shown in Figure 6.13 (b), is applied to the multipliers of the proposed accelerator. In

more detail, the NMED of the high-depth layers of Squeezenet, when the introduced approximation in this step is applied to all the multipliers, is higher than that of the approximation technique in the previous step. However, the NMED of mid-depth layers is lower when the introduced approximation in this step is applied to all the multipliers compared to that of the explored approximation technique in the previous step. Based on our evaluation the approximation method in this step compensates the error caused by the former approximation methods in the mid-depth layers.

The reason why the approximation method shown in Figure 6.13 (b) is not effective for the high-depth layers, is mostly shift operation itself. As described in the previous step, a few least significant bits of the multiplicand operands are replaced with an inexact bit which its logical value is "1". Shifting this constant logical value to columns with a greater significace increases the amount of error for some output features mostly in the high-depth layers. For the mid-depth layers shown in Figure 6.4, the introduced approximation in this step is effective because the magnitude of the ifmaps is such a way that after performing a shift operation, the magnitude of the 2 least partial product rows decreases as explained at the beginning of this Subsection. In more detail, by selecting $j - 1$ as 7, the magnitude of those partial product rows decreases after shift operation for the 2 least partial product rows. In other words, applying approximation shown in Figure 6.13 (b) compensates the mentioned error above caused by shift operation itself in mid-depth layers like Fire 4 and Fire 6.

However, for the high-depth layers where the magnitude of the ifmaps decreases, applying this approximation while performing shift operation would not compensate the error caused by shift operation itself. In fact, the shift operation still should be applied in an exact way for some ifmaps and should not be performed for some others while performing the computations of high-depth layers. For example, for the ifmaps that their magnitude is very small, shifting the constant logical value "1" from column 4 to 5 increases the computed NMED of these layers.

As shown in Figure 6.4, when the approximation which its dot diagram is shown in Figure 6.10 (b) is applied to the multipliers of the proposed accelerator, the NMED of the layers of SqueezeNet up to layer Fire 2, is the lowest. Furthermore, as explained above, for the mid-depth layers such as Fire 4 and 6, applying the approximation introduced in this step while performing shift operations decreases the NMED of these layers. In addition, for the high-depth layers, the shift

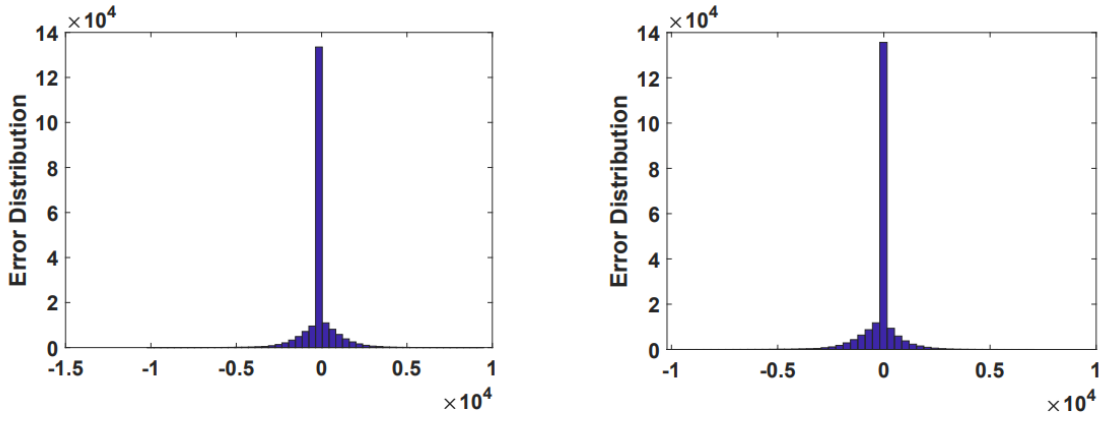


Figure 6.14: The error distribution of Layer Fire 4 when (a) the fixed version of the approximation introduced in step 4 (A), and when (b) the dynamic version of the approximation introduced in step 4 (B) is applied to all the Booth multipliers of the proposed accelerator.

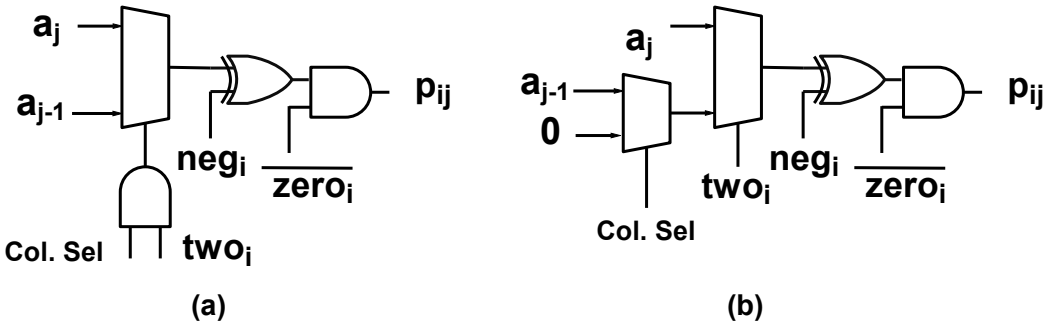


Figure 6.15: The gate-level architecture of the approximate partial product generator (a) which can avoid shift operation dynamically, and (b) which can coordinate to shift a specific logical value dynamically.

operation should be performed for some ifmaps and should not be performed for others, depending on the magnitude of the ifmaps.

**Step 4 (B)**

To apply various degree of the approximation introduced in step 4 (A), it is not possible to use a fixed architecture. A fixed version of the design shown in Figure 6.13 (b) comes with reducing area and power consumption [9]. However, as discussed, to have a higher chance to compensate

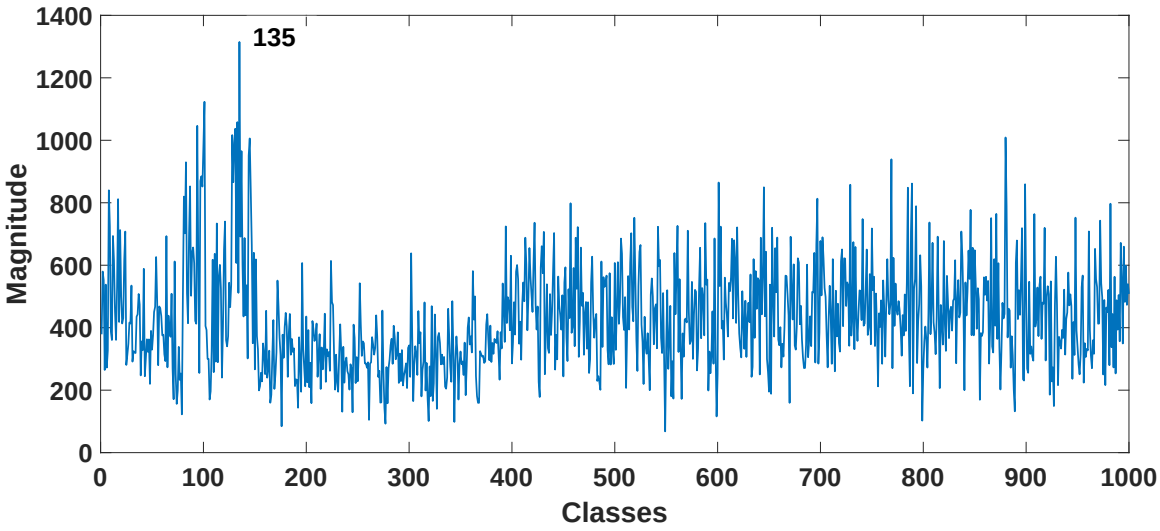


Figure 6.16: The output of the proposed accelerator, when a picture of a Crane is fed to this accelerator, and the dynamic version of the approximation introduced in step 4 (B), is applied to all the Booth multipliers in excess of the applied approximation in the previous steps.

the error of previous steps while running different DNNs, a dynamic version of this approximation technique should be implemented.

To perform the shift operation in an exact mode for the layers of SqueezeNet up to Fire 2, and to decrease the computed NMED of the mid-depth layers, the approximate partial product generators shown in Figure 6.15 (a) is proposed. As shown in Figure 6.15 (a), the signal $two_i$ in the partial product generator is gated for the partial products in columns 5 and 7 through some control signals. To add more detail, using this control signal, it is possible to apply the approximation introduced in this step whenever it is needed. For the high-depth layers, replacing the constant logical value "1" with "0" in column 4 avoids shifting the logical value "1" from column 4 to 5 for the small feature maps. In other words, it is like applying the truncation method evaluated in step 2 of this Subsection for some ifmpas. It is also possible to use the approximate partial product generator shown in Figure 6.15 (b) for the partial product generators in column 5 to compensate the error in high-depth layers in a more accurate way. To give more detail, the approximate partial product generator in Figure 6.15 (b), sets the logical value of the partial products in column 5 of the 2 least significant partial product rows to a specific value.

The ifmap distributor unit of the proposed accelerator which is explained in Subsection 6.1.3,

coordinates applying various degrees of this approximation in columns 5 and 7 for different layers. Furthermore, this unit coordinates the value of constant logical value in the fourth column while performing a shift operation. To give more detail, this unit coordinates the select control signal of the fifth column automatically based on the logical value of the bits $a_3$ to $a_0$. For example, in the lest-significant row, when the partial product is positive and the logical values of these bits are "11xx", then the constant logical value "1" will be shifted to column 5 like the exact mode but when the logical values of these bits are "0001", the constant logical value "0" will be shifted to column 5.

Figure 6.16 shows the output of the proposed accelerator when the same Crane picture used for bench-marking former approximate architectures, is used as an input of the accelerator and the dynamic version of the approximation technique shown in Figure 6.13 (b) is applied to all the multipliers of this accelerator. The computed probability using the Softmax function provided by Matlab for index 135, is 99% whihc is higher than that of the previous designs. Figure 6.14 (b) shows the error distribution of layer Fire 4 of the Squeezenet when the introduced dynamic approximation in this step, is applied to all the Booth multipliers, and the Crane picture shown in Figure 6.3 (a) is fed to the proposed accelerator. This distribution is balanced around zero compared to that of previous methods. The variance of this distribution is $0.83e+03$ which is the lowest than that of other explored designs. Furthermore, for almost all the layers with different number of channels the error distribution is balanced. Figure 6.4 shows the NMED of a few layers of Squeezenet when the dynamic version of the approximation techniques shown in Figure 6.13 (b), is applied to the multipliers of the proposed accelerator. In more detail, the average NMED of all the layers is the lowest than that of the previous steps. Moreover, the propagated error remains low through the DNN compared to previous techniques.

In this chapter, the approximation techniques targeting the accumulation stages of the multipliers are not evaluated. There are many kinds of approximate half-adders, full-adders, 4 to 2 compressors. However, the function of a few of these designs was evaluated while running DNNs [79,88]. In [79] the authors proposed two 4 to 2 compressors, each of them can compensate the error caused by the other type. They also built up two types of multiplier upon these compressors. In more detail, authors in [79], perform the computations of a specific number of the channels of each layer with

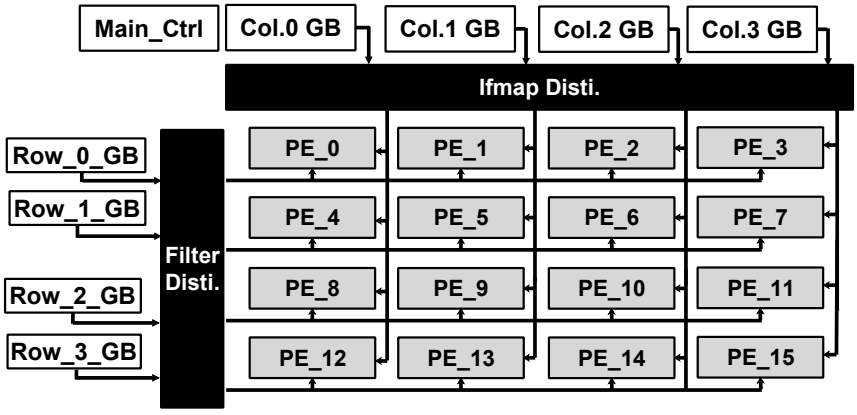


Figure 6.17: The general architecture of the proposed accelerator.

multiplier type 1 and they perform the rest of the computations of that layer with multiplier type 2. The proposed designs in [79] are simulated and discussed in Subsection6.1.4 of this chapter.

### 6.1.3 Proposed Architecture

The general architecture of the proposed accelerator is shown in Figure 6.17. The proposed computation engine contains 16 PEs, 4 column Global Buffers (GBs), 4 row GBs, an ifmap and a filter distributor units. Each regular PE contains 3 MAC units. Each column of the PEs computes an output feature of 4 different filters.

#### MAC Unit & PE Specifications

A MAC unit is built upon the re-configurable proposed Booth multiplier introduced in part B of step 4 of Subsection 6.1.2. Figure 6.18 (a) shows the internal architecture of this MAC unit. In each MAC unit, there is a re-configurable Booth multiplier, an adder, an accumulator, and an internal control unit. The bit widths of both the input ifmap and the filter weight is 8-bit. In the Booth multiplier, the approximation method explored in step 1 is already applied in a fixed format, since we believe the error regarding this approximation can be compensated using other approximate techniques. The proposed MAC unit supports all the approximation techniques explored in Subsection 6.1.2. Signal Trunc./Const1 is considered for replacing the constant logical value "1" with "0" in column 4 of the partial products. In short, using this signal, it is possible to apply the truncation method which is explained in step 2, as well as the evaluated approximation

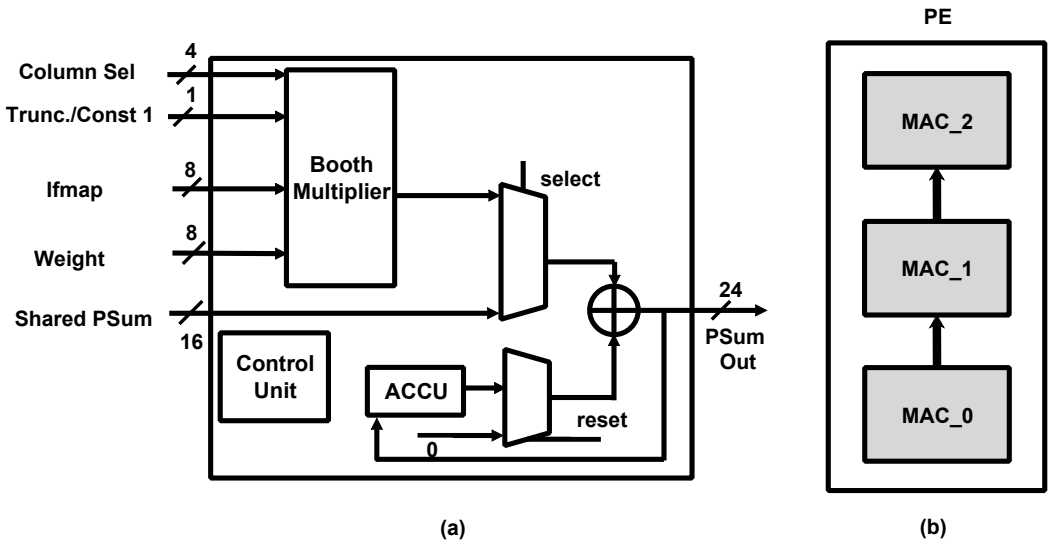


Figure 6.18: The internal architecture of (a) the proposed re-configurable MAC unit that supports all the approximation methods explored in Subsection 6.1.2 dynamically, and (b) each PE inside the computation engine.

technique in step 3. Signal Column Sel is considered for applying the dynamic format of the approximation technique explored in part B of step 4, while performing the MAC operations of different layers. the Shared PSum port in each MAC unit, is considered for accumulating the partial sums computed by other MAC units within each PE. Each MAC unit has a control unit that coordinates the computation process within that MAC unit.

Figure 6.18 (b) shows the internal architecture of each PE in the proposed accelerator. To give more detail, each PE includes 3 MAC units, and the computed partial sum by each MAC unit could be an input for another MAC unit within each PE. Accordingly, the ifmap and filter rows are divided into rows containing 3 elements before mapping into the computation engine. Zero padding is applied wherever it is needed.

### Distributor Units

The internal architecture of the ifmap distributor unit is shown in Figure 6.19. There are 3 main sub-units inside each distributor unit. These sub-units are a comparator, a scratchpad, and an internal control unit. The distributor unit can load up to 4 ifmaps concurrently from each

of the Column GBs into the scratchpad unit. The comparator unit considered for each column evaluates the logical value of the bits $a_3$ to $a_0$ of the corresponding ifmaps of that column. The internal control unit, based on the feedback it receives from the comparator units, coordinates the logical value of the signals Column Sel, Trunc./Const1 considered for the MAC units in each column. Furthermore, the control unit inside the distributor unit applies the dynamic format of the approximation techniques evaluated in step 4 to various columns of the computation engine. In addition, the ifmap distributor unit, in each cycle, can read four ifmaps from Column0 to Column3 GBs concurrently. This process continues based on the considered dataflow for performing the computations of each layer. Then, this unit feeds the PEs inside the computation engine with the Booth-encoded format of the loaded ifmaps. Moreover, the scratchpad unit in the ifmap distributor unit contains 4 ifmap sub-units. Each ifmap scratchpad sub-unit contains 16 8-bit registers to store the elements of the encoded ifmaps.

The internal architecture of the filter distributor units is somehow similar to ifmap distributor unit. However, there are no Comparator units in the filter distributor unit. Accordingly, there are no Trunc./ Cosnt1 and Column Sel signals. In addition, the control unit inside each distributor unit, coordinates the write and read data process into and from the mentioned scratchpads above. Furthermore, the main control unit starts each processing cycle by notifying the control unit inside the ifmap and filter distributor units to load their first stored ifmaps or encoded weights. Then, the main control unit notifies the internal control unit of the ifmap distributor unit to load all the remained ifmaps into the PEs in each cycle. The main control unit repeats this process for the remained encoded weights inside the filter distributor units.

**Global Buffers**

Each column GB contains 4 1.5kB SRAM buffers for keeping the elements of ifmap rows and 1.5kB SRAM buffers for keeping the partial sum values. Each row GB contains 3 1.5kB SRAM buffers for keeping the elements of each row of the filters. There are 4 small adder trees in each column GB that are considered to accumulate the computed partial sums by PEs to compute the final partial sums. Furthermore, 4 small control units in each column GB coordinate the write or read processes into or from the partial sum buffers. These units also coordinate the computations

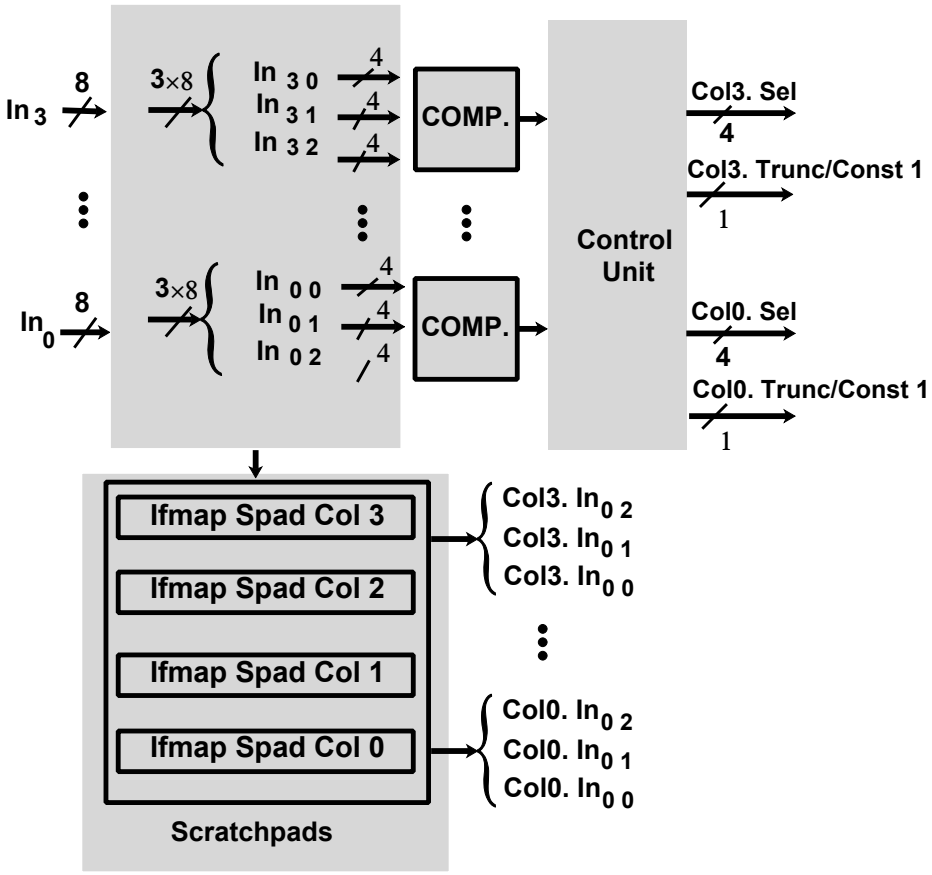


Figure 6.19: The internal architecture of the ifmap distributor unit of the proposed accelerator.

of the pooling layers.

### 6.1.4 Experimental Results and Discussion

#### Accuracy Analysis

To evaluate the accuracy of the explored multipliers in this Subsection, a cycle-accurate simulator is implemented in C#. This simulator is implemented based on the RTL design of the proposed accelerator explained in Subsection 6.1.3. Two image classifiers, namely SqueezeNet [7] and GoogleNet [72], and a speech command detector provided by Matlab [89], are used for evaluating the proposed architecture. The main reason for choosing these classifiers is to evaluate almost all different layer types of DNNs, and DNNs both with low and high number of layers. Also, exploring the effectiveness of the proposed approximation method on different applications. ImageNet ILSVRC2012 validation dataset [8] is used for evaluating the image classifiers and Google speech

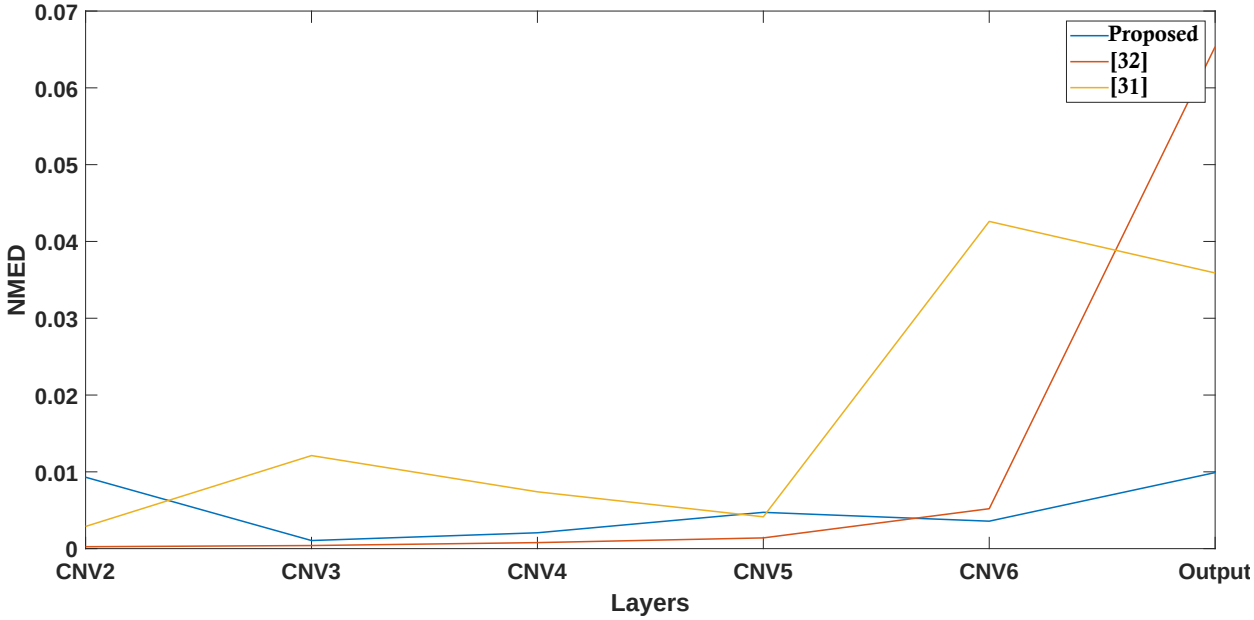


Figure 6.20: The computed NMED of a few layers of the speech command detector network for different approximate multipliers.

commands dataset [90] is used for evaluating the speech command detector. As the goal of this research is evaluating the effect of approximation while doing inference of DNNs, pre-trained DNN coefficients provided by Mat lab Deep Learning Tool are quantized into 8 bits and also sorted into rows containing 3 elements.

As authors showed in [86, 87], logarithmic multiplication is a promising approach for applying approximation into inference computations of DNNs. These types of multipliers converts the multiplication operations into additions by converting the inputs to the logarithm domain. These types of multipliers are efficient in terms of area and power consumption. Accordingly, two competitive logarithm-based designs which already have shown promising accuracy while performing inference of DNNs, are selected for comparison.

Figures 6.20 to 6.22 demonstrate the NMED of different layers of the benchmark DNNs when the exact multipliers in the proposed accelerator are replaced with various approximate multipliers. Due to the large number of layers of SqueezeNet and GoogleNet only the computed NMED of a few layers of these benchmark networks is shown in these figures. The given name to the layers of these DNNs, is the same as the given name in Matlab Deep Learning Toolbox. As shown

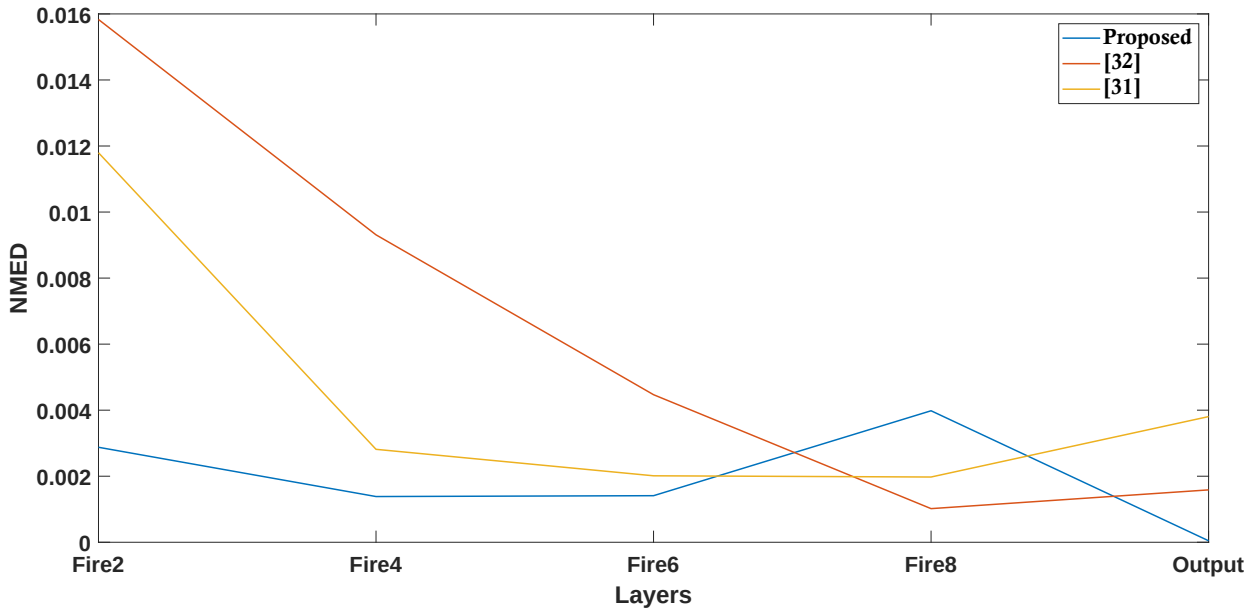


Figure 6.21: The computed NMED of a few layers of SqueezeNet for different approximate multipliers.

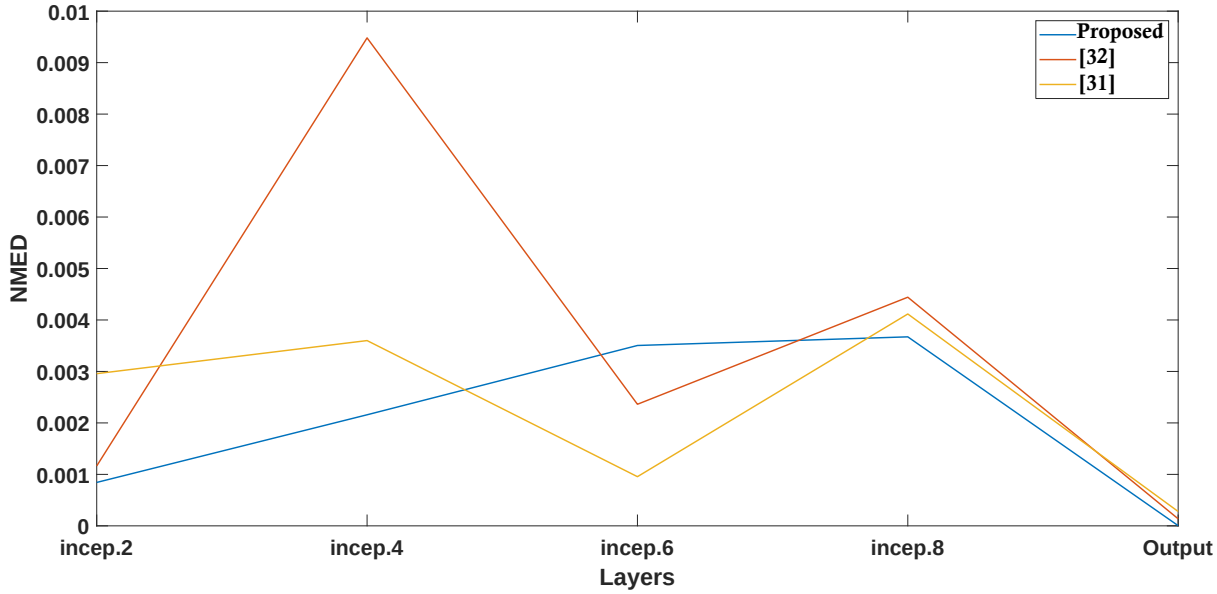


Figure 6.22: The computed NMED of a few layers of GoogleNet for different approximate multipliers.

in these figures, when the exact multipliers are replaced with the proposed Booth multiplier, the NMED of all the output layers of the benchmark DNNs, is lower than that of other approximate

Table 6.2: Accuracy report of the explored approximate designs while running benchmark DNNs

| DNN | Exact | [86] | [87] | Proposed |
|---|---|---|---|---|
| Speech Command Detector | 79.3 | 77.1 | 77.7 | 78.4 |
| SqueezeNet | 57.5 | 54.9 | 55.8 | 56.5 |
| GoogleNet | 68.7 | 66.7 | 67.1 | 67.6 |

designs. Accordingly, when the exact multipliers are replaced with the proposed Booth multiplier, the inference accuracy is higher compared to the computed inference accuracy of other approximate designs, while running benchmark DNNs.

Table 6.2 shows the accuracy of the benchmark DNNs when exact multipliers in the proposed accelerator are replaced with approximate multipliers. In general, the function of the proposed approximate multiplier is better when the number of MAC operations that have to be done regarding an output feature increases. To give more detail, when the number of MAC operations increases, the chance of having error in both positive and negative direction increases as well, which eventually ends up in a smaller accumulated error. Also, the 4 to 2 compressors proposed in [79] have been simulated and applied into the accumulation stage of the 8 least significant columns of the exact Booth multiplier. To add more information, the computations of the first half of the channels of each layer are performed using the first proposed multiplier type and the second half of the channels of each layer using the second proposed multiplier type in [79]. However, the inference accuracy drops sharply while running SqueezeNet and GoogleNet. The main reason is that the Booth algorithm reduces the number of partial product rows. Accordingly, there is only room to apply this approximation into the 4 to 2 compressors of columns 5 to 7 of the exact Booth multiplier. To add more detail, the chance of applying this approximation into low significant columns of an 8×8 radix-4 Booth multiplier is low.

### 6.1.5 Hardware Implementation

Table 6.3: Synthesis report of the explored multipliers.

| Multiplier | Area ($\mu m^2$) | Power ($mw$) | Critical Path ($ns$) |
|---|---|---|---|
| Exact | 844.5 | 0.310 | 1.29 |
| [86] | 812.4 | 0.240 | 1.13 |
| [87] | 781 | 0.253 | 1.51 |
| Proposed | 670 | 0.212 | 1.11 |

All the evaluated multipliers are implemented in Verilog and are synthesized using SYNOPSYS Design Compiler and TSMC 65nm technology library. ARM Artisan 65nm SRAM and register file generator are used for creating the global buffer and scratchpads. To measure the power consumption precisely, the activity of the signals inside the computation engine is stored in a saif file during the functional simulation of the design. Then, the design is synthesized using Design Compiler and used the mentioned saif file to report the power of each unit and its sub-units.

Table 6.3 shows the synthesis report of the evaluated multipliers in this Subsection. The power consumption of all the multipliers is measured when the working frequency is set to 200 Mhz for all the multipliers. When compared to the exact multiplier, the proposed design is $\times$ 1.26 smaller and $\times$ 1.46 more energy efficient. The same authors in [87], implemented an efficient version of this multiplier in [32] which is very optimized both in terms of area and energy consumption. The area of their proposed approximate multiplier in [32] is slightly smaller than our proposed design, and its power consumption is almost the same as that of our design. However, since they apply approximation in both the logarithm transform process and the logarithm domain addition process, their design is less accurate compared to the proposed design in [87].

A few accelerators which their general architecture is shown in Figure 6.17, are built up upon the explored multipliers, and their specifications are reported in Table 6.3. In accelerator 4, which is built up upon the proposed Booth multiplier, regardless of the multipliers, the ifmap distributor unit is also somehow different compared to other designs. To give more detail. The ifmap distributor

Table 6.4: Synthesis report of the explored DNN accelerators

| Hardware Spec. | Accelerator 1 | Accelerator 2 | Accelerator 3 | Accelerator 4 |
|---|---|---|---|---|
| Mult. Type | Exact | [86] | [87] | Proposed |
| Frequency | 200 | 200 | 200 | 200 |
| Core Voltage/Tech. | 1.1v/65nm | 1.1v/65nm | 1.1v/65nm | 1.1v/65nm |
| Reg. File | 0.125kB | 0.125kB | 0.125kB | 0.125kB |
| Comp. Engine Area | 178K ($\mu m^2$) | 173K ($\mu m^2$) | 168K ($\mu m^2$) | 150K ($\mu m^2$) |
| Comp. Engine Power | 32.3 ($mw$) | 28.8 ($mw$) | 27.5 ($mw$) | 25.2 ($mw$) |

unit in other designs does not include any comparator units. The size of the used Global Buffer is equal in all the developed accelerators. Therefore, to have a better understanding of the cost of different approximate multipliers, the power consumption and area of the Global Buffers are not reported in Table 6.4.

Although the area and power consumption of the ifmap distributor unit in accelerator 4 are higher, however, as shown in Table 6.4, the total area and power consumption of this accelerator is smaller than that of other designs. To give more detail, the proposed design achieves an area efficiency of $\times$ 1.19 and energy efficiency of $\times$ 1.28 compared to the exact design while running benchmark DNNs. The inference processing delay for all the developed accelerators while running different benchmark networks is the same. The inference processing delay while running the speech command detector network, SqueezeNet, and GoogleNet is 2.3$ms$, 25.1$ms$, and 86$ms$ respectively.

It is also possible to improve the area and energy efficiency of the proposed multiplier by sharing the Booth encoder part of the multipliers over all the multipliers in the same row of the computation engine. Therefore, instead of sharing weights, the filter distributor unit shares the signals $neg_i$, $two_i$, and $zero_i$ regarding each Booth-encoded element of each weight. Sharing part of the multiplier is not possible in every design, and it is challenging for some others. For example, in the evaluated logarithm-based multiplier in [87], the critical path is long, and sharing the encoder part comes with increasing the delay of the critical path more.

## 6.2 Applying Various Degrees of Approximation to Different layers of DNNs

### 6.2.1 Introduction

Although Intel and Xilinx provided hard cores such as DSP48 as a fast and configurable MAC unit in their FPGA devices, but when it comes to dealing with restrictions such as energy consumption, routing delay, and optimum utilization of the available resources, it is not always possible to map the MAC units of the developed design to DSP48 for all the applications. That is why Xilinx and Intel also provided soft multipliers based on look-up tables (LUTs). Applying approximation is an approach to optimize LUT-based MAC units in terms of resource and energy consumption. However, not all those designs are evaluated while running inference computations [91, 92]. In addition, all those proposed designs until now, only applied a specific type of approximation when performing the whole inference computations. In this paper, the tolerance of different layers of DNNs to various degrees of approximation is explored by changing the configuration of the FPGA resources dynamically at run-time.

Technically, the developers customize and optimize the design to decrease the logic counts in ASIC-based designs. However, in FPGA-based designs, re-configurable logic elements inside FPGA have a fixed number of inputs and outputs such as LUTs. Accordingly, it is not easy to fully control their utilization. As mentioned, Xilinx and Intel FPGAs provide fast DSP48 hardcore multipliers for digital signal processing applications. However, using these hard cores can incur a large routing delay since they are available at some locations on an FPGA. Furthermore, DSP48 can perform other mathematical operations such as division. Therefore, they include extra logic which make them more expensive in terms of static power consumption. As an example, the power consumption of a DSP48 while performing a multiplication operation is higher than a LUT-based multipliers especially when the bit-width of the inputs is less than 16. LUTs are distributed all over the FPGA platform, which can makes the placement and routing optimum in terms of routing delay. In addition, depending on the device, the number of DSP48 cores is limited and if an application uses all the hard DSP48 cores, the only way to implement the rest of the design is through using LUTs. All the aforementioned notes highlight that optimizing the LUT-based designs is necessary. In what follows, the tolerance of different layers of DNNs to various degrees of approximation is explored.

### 6.2.2 Applying Various Degrees of Approximation at Run-Time

Although, the authors in [93, 94] claim that applying approximation into high-depth layers in DNNs has more effect on the inference accuracy compared to earlier layers, but authors in [52] reported that applying approximation into fully connected layers almost does not have any effect on inference accuracy. Accordingly, there is a need to investigate the tolerance of different layers of DNNs to approximation. To the best of our knowledge, all the proposed designs until now, have used only a specific type of approximation while performing the MAC operations of each inference. As discussed in Subsection 6.1.3, applying approximation to higher significant columns of the multipliers causes a sharp drop in inference accuracy. However, applying more approximation to multipliers comes with less energy consumption and resource utilization. Therefore, it is still motivating to investigate the degree of approximation that can be applied to the computation engine of DNN accelerators while performing the MAC operations of some layers of DNNs.

DNNs can work with smaller data types such as 8-bit integers which makes them suitable for integration into edge devices [80]. In other words, it is possible to achieve an acceptable accuracy by utilizing 8-bit integer multipliers instead of regular 32-bit floating-point multipliers. Radix-4 Booth multiplication is a popular multiplication algorithm that reduces the size of the partial product array by half. To investigate the tolerance of different layers of DNNs, two 8-bit approximate radix-4 signed Booth multipliers are developed. These multipliers are shown in Figure 6.23. As the considered bit-width of the multiplier operand is 8, the radix-4 Booth encoded of each weight will have 4 elements. In Figure 6.23, the partial product bits are organized into 4 rows and 16 columns. In this figure, the numbers given to the rows and columns of partial products are based on their significance.

In more detail, to apply various degrees of approximation, the considered re-configurable approximate 8×8 radix-4 Booth multipliers shown in Figure 6.23 (a) and (b) are proposed. These designs are inspired by the approximate Booth multiplier in [9]. The difference between the multipliers shown in Figure 6.23 is that OR gates are used to perform the accumulation of the partial product bits in columns 4 to 6 in Figure 6.23 (b) compared to Figure 6.23 (a). In more detail, the approximate multiplier in Figure 6.23 (b) is a version of the approximate multiplier shown in Figure 6.23 (a). The results reported in Subsection 6.2.4 confirm that the approximate multiplier

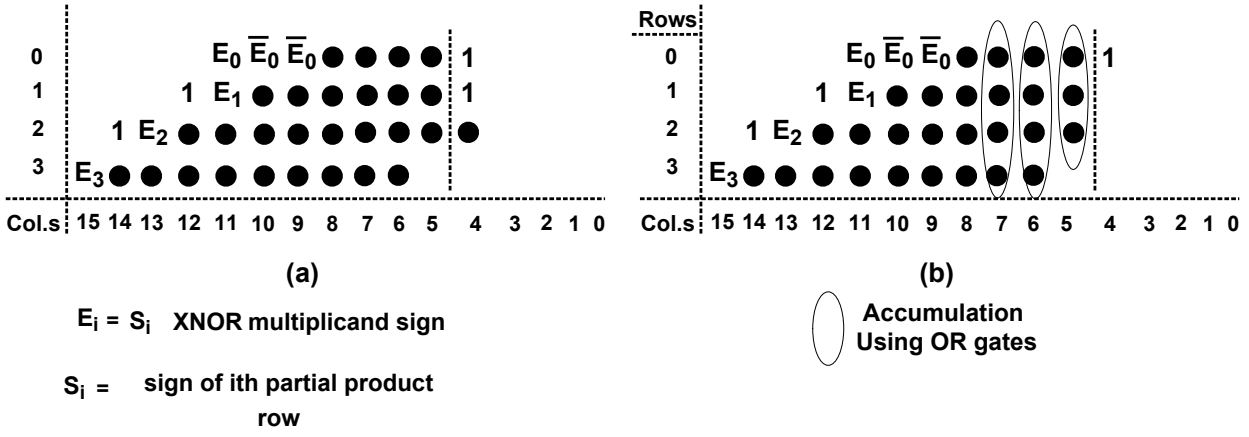


Figure 6.23: The dot-diagram of an approximate signed radix-4 Booth multiplier when (a) approximate partial product generators in the columns 0 to 4 of the 2 least significant rows are replaced with a constant "1" logical value, and when (b) partial product bits of the in the columns of 4 to 7 of the design shown in (a) are accumulated using OR gates. ● : exact partial products.

in 6.23 (b) is more optimized in terms of resource utilization and energy consumption. In other words, to apply various degrees of approximation, the approximate multipliers shown in 6.23 (a) and (b) interchangeably to perform low-depth layers and high-depth layers of a DNN and evaluate how the accuracy of that specific DNN changes.

**Case Study 1, Applying Various Degree of Approximation into an Environment Sound Classifier**

An Environment Sound Classifier (ESC) is developed and trained using the Matlab deep learning toolbox. The used data-set is ESC-10 [95]. The audio feature extractor function provided by Matlab is used to extract the features of the input sounds. The internal architecture of this sound classifier includes 3 convolutional layers and 3 fully-connected layers. The confusion matrix of this classifier for validation data-set using 32bit floating-point weights and Matlab deep learning toolbox is shown in Figure 6.24. When the approximate multiplier is shown in Figure 6.23 (a) is used to perform all the MAC operations of all the layers of the mentioned environment sound classifier, the inference accuracy only drops by 1.25%. The confusion matrix of the environment sound classifier when the approximate multiplier shown in Figure 6.23 (a) is used to perform all the computations, is shown in Figure 6.25. To evaluate the effect of applying various degree of

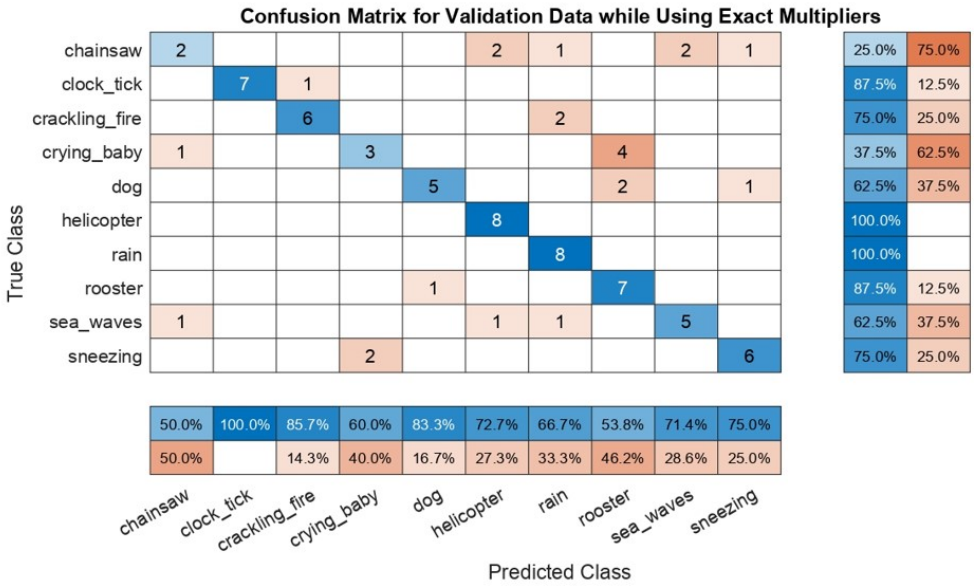


Figure 6.24: The inference accuracy of the environment sound classifier computed by Matlab which uses pre-trained 32bit floating-point weights to perform the inference

approximation, two scenarios are considered. In the first scenario, the approximate multiplier shown in Figure 6.23 (b) is used to perform the MAC operations of the first 3 convolutional layers and the approximate multiplier shown in Figure 6.23 (a) is used to perform the MAC operations of the rest of the layers. Meaning a higher degree of approximation is applied to low-depth layers compared to high-depth layers. The confusion matrix of the first scenario for validation data is shown in Figure 6.26. It is clear that although more approximation is applied to the earlier layers, the inference accuracy is almost the same as the confusion matrix shown in Figure 6.25. In the second scenario, the approximate multiplier shown in Figure 6.23 (a) is used to perform the MAC operations of the first 3 convolutional layers and the approximate multiplier shown in Figure 6.23 (b) is used to perform the MAC operations of the rest of the layers. Meaning a lower degree of approximation is applied into low-depth layers and a higher degree of approximation is applied to high-depth layers. The inference accuracy in the second scenario drops 4%. Meaning a smaller degree of approximation should be applied to the high-depth layers compared to low-depth layers of the developed environment sound classifier when running inference computations.

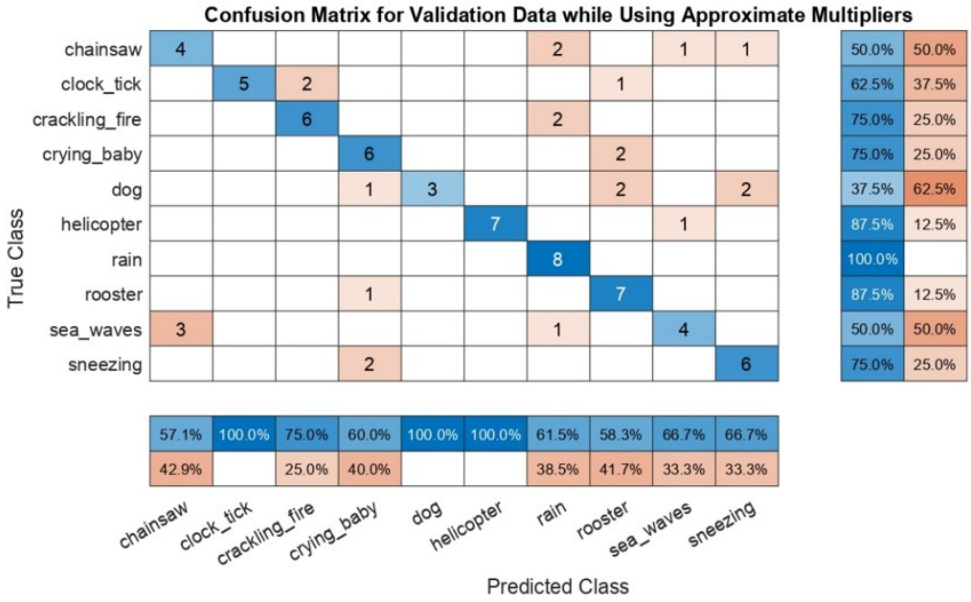


Figure 6.25: The inference accuracy of the environment sound classifier when approximate 8-bit Booth multiplier shown in Figure 6.23 (a) is used to perform the inference computations

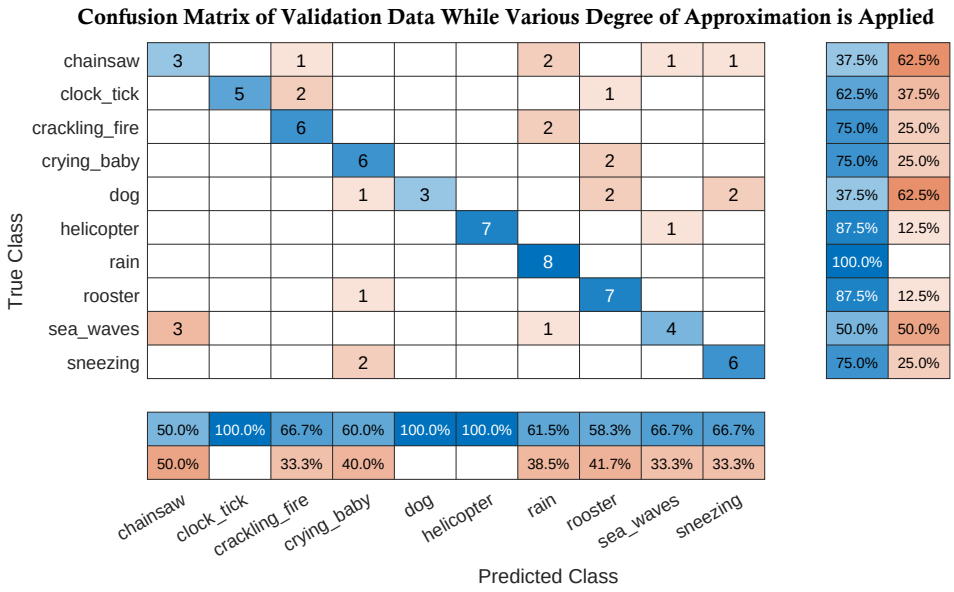


Figure 6.26: The inference accuracy of the environment sound classifier when approximate 8-bit Booth multiplier shown in Figure 6.23 (b) is used to perform the inference computations

### Case Study 2, Applying Various Degree of Approximation into ResNet-20

In the second case study, a pre-trained ResNet-20 is explored. The used data-set is CIFAR-10 [96]. This image classifier has 20 convolutional layers and 1 fully connected layer. The validation

**(a)** **(b)**

Figure 6.27: The picture of (a) a Crane vs (b) an American Egret

accuracy of ResNet-20 while using exact 8-bit multipliers is 88%. Like the former case study, two scenarios are considered for applying various degrees of approximation while performing the MAC operations of this image classifier. In the first scenario, the approximate multiplier shown in Figure 6.23 (b) is used to perform the MAC operations of the first 10 convolutional layers and the approximate multiplier shown in Figure 6.23 (a) is used to perform the MAC operations of the rest of the layers. The inference accuracy is 76% for the first scenario. In the second scenario, the approximate multiplier shown in Figure 6.23 (a) is used to perform the MAC operations of the first 10 convolutional layers and the approximate multiplier shown in Figure 6.23 (b) is used to perform the MAC operations of the rest of the layers. The inference accuracy for the second scenario is 79.2%. Accordingly, a smaller degree of approximation should be applied to the low-depth layers compared to high-depth layers of ResNet-20 classifier when running inference computations.

The behavior of the selected DNN in the first and second case study is different while applying various degrees of approximation which even shows the value of this research more. Technically, the low-depth layers detect high-level concepts and high-depth layers detect more details. In the first scenario, the picture is shown in Figure 6.27 (a) is misclassified as an American Egret which its picture is shown in Figure 6.27 (b). As mentioned earlier, low-depth layers in the ResNet-20 detect high-level concepts like the form of the head, beak, tail, and legs. To give more detail, although by using the multiplier shown in Figure 6.23 (b), it is still possible to detect the mentioned high-level concepts, However, applying more approximation in the low-depth layers of ResNet-20 removes the details which eventually causes a misclassification. However, in the second scenario,

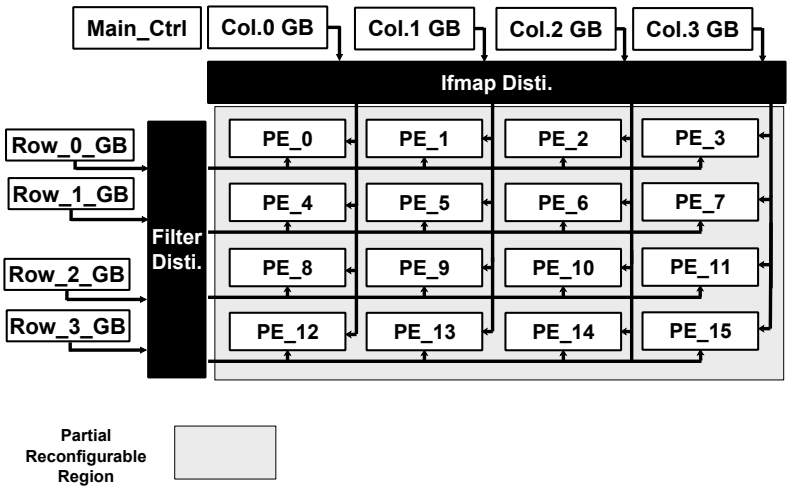


Figure 6.28: The general architecture of the proposed accelerator.

the picture shown in Figure 6.27 (a) is classified correctly as a Crane. In what follows we discuss the proposed re-configurable computation engine suitable based on FPGA platforms suitable for applying various degrees of approximation into different layers of DNNs.

### 6.2.3 Proposed Architecture

The general architecture of the proposed accelerator is shown in Figure 6.28. This architecture is similar to the proposed architecture in the Subsection 6.1.3 of Section 6.1. To add more detail, in this section the target platform is FPGA. To apply various degrees of approximation dynamically at run-time, the MAC units are mapped into a partial re-configurable block. The proposed computation engine contains 16 PEs, 4 column GBs, 4 row GBs, an ifmap, and a filter distributor unit. Each regular PE contains 9 MAC units. Each column of the PEs computes an output feature of 4 different filters.

#### MAC Unit & PE Specifications

The internal architectures of the proposed MAC unit and PE are shown in Figures 6.29 and 6.30. In each MAC unit, there is an approximate multiplier, an accumulator, and an internal control unit. The bit-width of both the input ifmap and the filter weight is 8-bit. To verify the results concluded in Section 6.2.2 while using the proposed multipliers, the approximate multipliers from previous

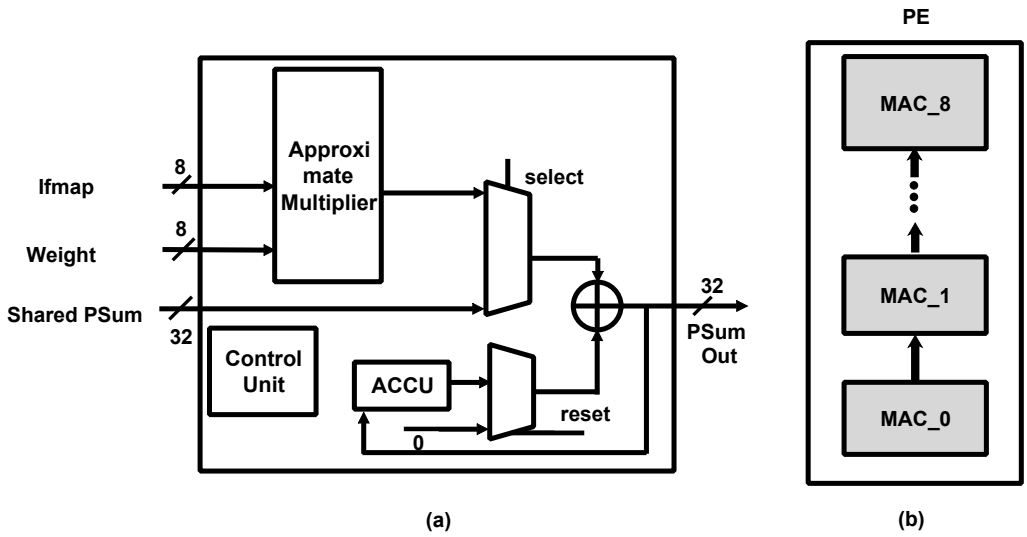


Figure 6.29: The internal architecture of (a) the proposed MAC unit which includes an approximate multiplier, and (b) each PE inside the computation engine.

works are evaluated [86, 87, 91, 97]. the Shared PSum port in each MAC unit is considered for accumulating the partial sums computed by other MAC units within each PE. Each MAC unit has a control unit that coordinates the computation process within that MAC unit.

Figure 6.29 (b) shows the internal architecture of each PE in the proposed accelerator. To give more detail, each PE includes 9 MAC units, and the computed partial sum by each MAC unit could be an input for another MAC unit within each PE. Accordingly, the ifmap and filter rows are divided into rows containing 3 elements before mapping into the computation engine. Zero padding is applied wherever it is needed.

### Distributor Units

The internal architecture of the ifmap distributor unit is shown in Figure 6.30. The distributor unit can load up to 4 ifmaps concurrently from each of the Column GBs into the scratchpad unit. In addition, the ifmap distributor unit, in each cycle, can read 3 ifmaps from Column0 to Column3 GBs scratchpads concurrently. This process continues based on the considered dataflow for performing the computations of each layer.

The internal architecture of the filter distributor units is somehow similar to ifmap distributor

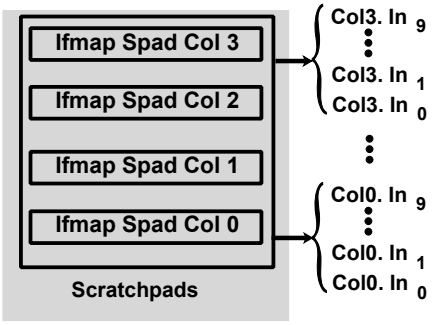


Figure 6.30: The internal architecture of the ifmap distributor unit of the proposed accelerator.

unit. In addition, the control unit inside each distributor unit coordinates the write and read data process into and from the mentioned scratchpads above. Furthermore, the main control unit starts each processing cycle by notifying the control unit inside the ifmap and filter distributor units to load their first stored ifmaps or encoded weights. Then, the main control unit notifies the internal control unit of the ifmap distributor unit to load all the remained ifmaps into the PEs in each cycle. The main control unit repeats this process for the remained encoded weights inside the filter distributor units.

#### Global Buffers

Each column GB contains two $16 \times 24bit$ Block RAMs for keeping the elements of ifmap rows and an $8 \times 32bit$ Block RAMs buffer for keeping the partial sum values. Each row GB contains a $16 \times 24bit$ for keeping the elements of each row of the filters. There are 4 small adder trees in each column GB that are considered to accumulate the computed partial sums by PEs to compute the final partial sums. Furthermore, 4 small control units in each column GB coordinate the write or read processes into or from the partial sum buffers. These units also coordinate the computations of the pooling layers.

### 6.2.4 Experimental Results and Discussion

#### Hardware Implementation

Table 6.5: Implementation report of the explored approximate computation engines.

| Used Multiplier in Computation Engine | LUT | FFs | Slack ($ns$) | Total on Chip Power ($w$) |
|---|---|---|---|---|
| Exact | 10244 | 181 | 0.69 | 1.3 |
| [97](2HH) | 6192 | 161 | 1.29 | 0.98 |
| [97](17KS) | 4367 | 125 | 1.49 | 0.84 |
| [97](JV3) | 2651 | 104 | 1.6 | 0.78 |
| [97](CK5) | 7117 | 159 | 1.083 | 1.027 |
| [97](2P7) | 6844 | 156 | 1.28 | 1.010 |
| [91] | 6579 | 139 | 1.2 | 1.009 |
| [87] | 10376 | 138 | 0.447 | 1.21 |
| [86] | 5445 | 138 | 0.684 | 0.910 |
| Mult.1[1] | 6396 | 164 | 1.003 | 0.983 |
| Mult.2[2] | 5939 | 162 | 1.34 | 0.960 |

[1]Is shown in Figure 6.23(a)

[2]Is shown in Figure 6.23(b)

A few multipliers are considered to confirm the observed behavior of DNNs in Subsection 6.2.2. All the evaluated multipliers are implemented in Verilog and are synthesized using Vivado. The target device is Zynq UltraScale+ XCZU7EV-2FFVC1156. The reported power is the total average on-chip power which includes both static and dynamic power. The static power of the FPGA device for all the designs is $616mw$. Table 6.5 shows the implementation report of the evaluated approximate multipliers. To add more detail, In all the designs, the computations engine includes $16 \times 9$ MAC units. The power consumption and the slack time is measured when the working frequency of the whole computation engine is set to 100 MHz for all the different designs.

**Accuracy & Performance Analysis**

To evaluate the accuracy, two pre-trained image classifiers, namely ResNet-20 [98] and Vgg-16 [15], a pre-trained environment sound classifier namely ESC, and a pre-trained digit recognition

network namely LeNet are considered. To run the image classifiers, CIFAR-10 [96] data-set is used. Also, ESC10 [95], and MNIST [99] data-sets are used to run the ESC and LeNet respectively. When the computation engine is re-configured in exact mode, the accuracy of LeNet, ESC, ResNet-20, and VGG-16 is 98.01%, 71.25%, 88%, and 90% respectively. Table 6.6 shows the accuracy of the benchmark DNNs when the exact computation engine in the proposed accelerator are replaced with approximate configurations. To investigate the effect of applying various degrees of approximation, for all the benchmarks, the layers of each specific DNN are grouped into two parts. For example, if a DNN has $n$ number of layers including both convectional and fully connected layers, the first $\lfloor \frac{n}{2} \rfloor$ layers are grouped as Part1 and the rest of the layers are grouped as Part2. Furthermore, two scenarios are considered while running the benchmark networks. In the first scenario, a more accurate multiplier is selected to perform the MAC operations of the layers grouped as Part1, and a less accurate multiplier is chosen to perform the MAC operations of the layers grouped as Part2. However, in the second scenario, a less accurate multiplier is chosen to perform the MAC operations of the layers grouped as Part1, and a more accurate multiplier is selected to perform the MAC operations of the layers grouped as Part2. For those multipliers that come with error in both directions compared to exact multipliers, when the number of MAC operations is high, the chance of having error in both positive and negative direction increases as well, which eventually ends up in a smaller accumulated error. The opposite is true when the number of MAC operations is low. In addition, the multipliers that do not come with the error in both directions are not suitable to perform the MAC operations in computation-intensive layers of DNNs.

Based on the results shown in Table 6.6, the achieved inference accuracy in the second scenario is higher than the first one for simple DNNs such as LeNet and the developed ESC. However, the achieved inference accuracy in the first scenario is higher than the second one for more complex DNNs such as ResNet-20 and VGG-16. As mentioned before, technically the layers grouped as Part1, detect high-level concepts and the layers grouped as Part2 detect more details. In LeNet and the developed ESC, successful detection of high-level concepts such as edges of the input image is enough to classify that input image correctly. To add more information, that it why these DNNs are smaller and their internal architecture is simpler than ResNet-20 or VGG-16. As shown in Table 6.6, it is possible to use less accurate multipliers to perform the computations of Part1 of

these DNNs to detect edges and high-level concepts. Furthermore, the number of MAC operations needed to be performed while running Part2 of LeNet and ESC is low, meaning more accurate multipliers are needed to perform the MAC operations of this part.

However, in ResNet-20 and VGG-16, regardless of the high-level concepts, more critical details available in the input image are needed to classify an input image correctly. To add more detail, although using less accurate multipliers to perform the computations of Part1 of ResNet-20 and VGG-16, still helps to detect high-level concepts inside an input image, however, it might cause the loss of some critical information needed for the correct classification of the input image eventually. As discussed in an example earlier, the picture of a Crane is classified as an American Egret while performing the MAC operations of low-depth layers of ResNet-20 using less accurate multipliers. To clarify more, there are some similarities between these two birds like the form of their neck, legs, and wings. However, there are some critical details that need to be transferred from low-depth layers to high-depth layers in order to classify the image correctly. This simple example confirms that for such complex DNNs, the critical details in the input image must be transferred from earlier layers to deeper layers while running inference of DNNs.

As shown in Table 6.6, the selected configurations for running different benchmarks are different, since not all the multipliers shown in Table 6.5 are suitable for running all the DNNs. Accordingly, except the considered configuration for running Vgg-16, all other configurations are selected in such a way as to achieve an acceptable inference accuracy for both scenarios while running the rest of benchmark DNNs. Based on our evaluation while running Vgg-16, using the available approximate designs in Table 6.5, it is not possible to achieve acceptable accuracy for both scenarios. This finding even highlights the benefits of both having a re-configurable computation engine and being able to apply various degrees of approximation more.

The architecture in Subsection 6.2.3 is proposed to investigate the tolerance of different layers of DNNs to approximation. The outcome of this research which is reported in Table 6.6 is expected to help designers to consider the behavior of different layers of DNNs while optimizing the computation engine of DNN accelerators using approximate computing techniques. Partial

Table 6.6: Accuracy report of the explored approximate computation engines

| Accu. Eval. | First Scenario | | | Second Scenario | | |
|---|---|---|---|---|---|---|
| Spec. | Part1 Mult. | Part2 Mult. | Accu.(%) | Part1 Mult. | Part2 Mult. | Accu.(%) |
| LeNet | Mult.1 [1] | Mult.2 [2] | 97 | Mult.2 | Mult.1 | 98.01 |
| | [87] | [86] | 94.21 | [86] | [87] | 95.65 |
| | [97](CK5) | [97](JV3) | 96.03 | [97](JV3) | [97](CK5) | 98.01 |
| | Exact | [97](2HH) | 97.8 | [97](2HH) | Exact | 98.02 |
| | [97](CK5) | [97](17KS) | 97.69 | [97](17KS) | [97](CK5) | 98.01 |
| ESC [3] | Mult.1 | Mult.2 | 66.1 | Mult.2 | Mult.1 | 70 |
| | Exact | [97](CK5) | 65 | Exact | [97](CK5) | 67 |
| | [97](CK5) | Mult.2 | 66 | Mult.2 | [97](CK5) | 70 |
| | Exact | Mult.1 | 69 | Mult.1 | Exact | 70 |
| | [91] | Mult.2 | 66.2 | Mult.2 | [91] | 70.2 |
| ResNet-20 | Mult.1 | Mult.2 | 79.2 | Mult.2 | Mult.1 | 76 |
| | Exact | [97](CK5) | 86.1 | [97](CK5) | Exact | 84.7 |
| | [97](CK5) | Mult.2 | 78 | Mult.2 | [97](CK5) | 76.7 |
| | Exact | Mult.1 | 86 | Mult.1 | Exact | 84.5 |
| | [91] | Mult.2 | 78 | Mult.2 | [91] | 76 |
| VGG-16 | Exact | Mult.1 | 88.6 | Mult.1 | Exact | 26 |
| | Exact | [91] | 88.3 | [91] | Exact | 40 |
| | Exact | [87] | 81 | [87] | Exact | 20 |
| | Exact | [97](CK5) | 82 | [97](CK5) | Exact | 16 |
| | Exact | [97](2P7) | 87 | [97](2P7) | Exact | 15 |

[1]Is shown in Figure 6.23(a)

[2]Is shown in Figure 6.23(b)

[3]Environment Sound Classifier

reconfiguration comes with run-time reconfiguration latency overhead [100]. It is worth mentioning that while evaluating the accuracy of the benchmark DNNs reported in Table 6.6, it is made sure that the configuration of the PEs is changed before starting the computations of the second part of DNNs. When loading bitstreams from internal storage, the time to load a region is a function of

the bitstream size, approximately 400 MB/sec on an Ultrascale+ device from Xilinx [100]. If there are enough resources available on FPGA to perform the computations of both low and high-depth layers concurrently, there is no need to dynamically reconfigure the architecture of PEs. However, if there are not enough resources available, depending on the inference delay, it is still possible to achieve energy efficiency using the proposed architecture in Subsection 6.2.3 while running DNNs.

Table 6.7, shows the achieved performance by accepting at most 2% loss in accuracy while running benchmark DNNs. For each specific DNN, the best approximate configurations of the computation engine in terms of both power consumption and inference accuracy are selected and explored. The inference per Joule is a fair metric to compare the performance of different computation engines. This metric means how many inferences are possible by consuming one Joule while running different benchmarks. The extra power consumption that comes while dynamically re-configuring the PES is considered as well [101]. The latency of dynamic reconfiguration of an array of four multipliers is 0.6 milliseconds. The inference delay of LeNet, ESC, ResNet-20, and VGG-16 is 4.4 ms, 9.4 ms, 4 ms, and 24 ms. For LeNet and ESC, there is only a need to change the configurations of a few multipliers since the second part of these DNNs includes only fully connected layers that come with a low chance of local data reuse. Accordingly, the configuration time is short for these DNNs. Considering the latency and the extra power consumption of the partial reconfiguration process, it is not possible to achieve considerable energy efficiency while running ResNet-20, and VGG-16. However, while running a larger format of VGG-16 which is pre-trained using the ImageNet dataset, the inference per Joule is $\times 1.27$ higher than the exact model. The inference delay of the proposed architecture while running this DNN is 3.2 Seconds.

Table 6.7: Performance report of the explored approximate computation engines while accepting 2% accuracy loss.

| Perf. Eval. | Exact | | | | Approx. | | | |
|---|---|---|---|---|---|---|---|---|
| DNN | LeNet | ESC | ResNet-20 | Vgg-16 [1] | LeNet | ESC | ResNet-20 | Vgg-16 |
| Part1 Mult. | Exact | | | | [97](17KS) | Mult.2 | Exact | Exact |
| Part2 Mult. | Exact | | | | [97](CK5) | [91] | Mult.1 | Mult.1 |
| Loss in Accu. | N/A | | | | 0% | 1.05% | 2% | 1.4% |
| Inference/j | 247 | 243 | 207 | 21 | 311 | 280 | 200 | 22.5 |

[1]Pre-trained VGG-16 using CFRA-10 dataset

# 7. Conclusions and Future Works

## 7.1 Summary & Conclusions

Equipping mobile platforms with deep learning applications such as healthcare services, speech processing, and real-time computer vision algorithms makes these platforms more valuable in remote areas. However, this is not feasible if the energy consumption barrier is not addressed. As an example, an estimation of the needed number of MAC operations per second for a self-driving car system is explored in Chapter 1. This chapter helps the readers understand the main motivation behind the proposed ideas in the dissertation which is designing efficient architectures for the acceleration of DNNs. In Chapter 2, different architectures proposed for acceleration of DNNs are discussed. In more detail, a unique set of categories are considered for classifying proposed contemporary architectures. Chapter 2 is expected to help the readers understand the main ideas behind the state-of-the-art accelerators until now.

In Chapter 3, a novel DNN accelerator for mobile platforms which reorders the computations based on the sign bit of the weights inside the computation engine is proposed. The proposed idea can be integrated into spatial DNN accelerators to enhance their performance further by cutting the computations earlier whenever the rest of the computations are ineffectual. Early cutting of the computations comes with fewer accesses to a higher level of memory which counts as a bottleneck for many contemporary works as well. When compared to [1], the proposed work improves throughput and energy efficiency respectively by $\times$2.34 and $\times$2.08 while running AlexNet, and similarly $\times$2.04 and $\times$1.81 while running VGG-16.

In Chapter 4, a real-time architecture is proposed for increasing the utilization of the PEs inside the spatial computation engine of DNN accelerators. The main idea is to use the available slack time caused by computation-pruning techniques and the used NoC format. When compared to

the reference design, the proposed work improves throughput and energy efficiency respectively by ×1.24 and ×1.18 while running AlexNet, VGG16, and MobileNet V2. The proposed NoC and connections between PEs are very efficient in terms of resource utilization. It is possible to use the proposed architecture to increase the utilization of the PEs in spatial DNN accelerators that are not equipped with an expensive NoC to speed up the computations.

To improve the performance of DNN accelerators, a real-time DNN accelerator that utilizes the chance of pruning identical effectual computations by finding identical bit values of corresponding ifmaps and filter weights is proposed in Chapter 5. To find out identical bit values and prune them, the ifmaps and filter weights are Booth-encoded, and also the MAC operations are decomposed down to bit level. The proposed architecture prunes the effectual computations by identifying identical elements of the encoded ifmaps and filter weights only once using the shared PEs. Shared PEs are a smaller version of regular PEs that are considered for performing the computations of identical elements. When compared to the reference design [27], the proposed work prunes the average GMAC per inference by ×1.16, ×1.17, ×1.16, ×1.16, and ×1.17 while running VGG16, GoogleNet, MobileNet V2, SqueezeNet, and Xception respectively. The hardware evaluation confirmed that by pruning the computations in the benchmark networks, the proposed design achieves an average speedup of ×1.4 and energy efficiency of ×1.21 per inference while maintaining the accuracy. It is possible to equip the binary, ternary, and all DNN accelerators that decompose the MAC operations to bit-level with the proposed architecture to enhance them in terms of performance.

In Chapter 6, applying various approximate techniques to the multipliers of DNN accelerators is studied experimentally. To give more detail, there are two sections in this chapter. The first section explains why applying approximation to DNNs is somehow different from other applications. Then, a step-wise approach for designing a re-configurable Booth multiplier is presented. And then, the effect of applying the introduced approximation in each step is experimentally studied. Furthermore, the effect of applying a combination of all the explored approximation techniques while running different layers of DNNs is studied. After that, a re-configurable multiplier architecture is proposed which supports all the approximation techniques explored in the proposed step-wise approach. Finally, a DNN accelerator is built upon the proposed multiplier and a few other competitive multipliers that are suitable for performing inference of DNNs. By accepting less that 2% loss in

accuracy, the proposed design achieves an area efficiency of $\times$ 1.19 and energy efficiency of $\times$ 1.28 compared to the exact design while running benchmark DNNs.

In the second section of Chapter 6, the tolerance of different layers of DNNs to approximation is explored. Firstly, it is explained why applying approximation to earlier layers is different compared to deeper layers. Then a re-configurable computation engine is proposed. After that, 10 different configurations were implemented. Then, the effect of applying various degrees of approximation to both simple and complex DNNs is evaluated and compared. This chapter is expected to help the readers understand the necessity of studying the error characteristics of approximation methods, and the behavior of total accumulated error when performing the computations of different layers of DNNs, before applying these methods to DNN accelerators. In addition, this chapter provides a better understanding of the tolerance of different layers of DNNs to approximation which helps the designers to achieve a reasonable trade-off between accuracy and energy efficiency.

## 7.2 Future Works

- There is a gap between what the DNN designers or data scientists perceive and what a hardware engineer implements to accelerate the computations of DNNs. Recently, quantization libraries are added into different platforms which brings a more accurate insight of the final implementation of accelerators to developers. However, research on how to provide instant feedback on the processing throughput and energy efficiency in the design or training of a DNN still seems vital.

- Quantization of the weights of DNNs based on the behavior of approximate multipliers remains as a future work. To add more detail, the error distance of an approximate multiplier varies for different inputs, depending on the internal architecture of approximate multipliers. There is no research on this topic and all the available approximate designs are evaluated using the pre-trained quantized weights.

- This dissertation offers useful insights into the special computational characteristics of deep learning inference, such as tolerance for approximate computing. It remains as a future work to investigate the integration of existing compression algorithms like weight sharing with

proposed approximate computing methods in this dissertation.

- A reconfigurable architecture comes with the chance of applying different approximate computing techniques. As mentioned in Chapter 6, logarithm based multipliers also come with a promising accuracy for some applications. However, it is still beneficial to investigate the inference accuracy by applying both exact domain approximation and logarithm domain approximation while performing the computations of different layers within a DNN.

- Investigation of the effect of applying various degrees of approximation into different convolutional layers of a DNN remains as a future work as well. For example, the Fire layer in MobileNet includes $1 \times 1$, $3 \times 3$, and $5 \times 5$ convolutional layers. In more detail, depending on the number of MAC operations, the degree of approximation that could be applied to each of the mentioned convolutional layers would be different.

# References


[1] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[2] "Google tpu v3 architecture," https://cloud.google.com/tpu/docs/system-architecture-tpu-vm, Updated 2022 (accessed March 26, 2022).

[3] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[5] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2017.

[6] S. Okumura, M. Yabuuchi, K. Hijioka, and K. Nose, "A ternary based bit scalable, 8.80 tops/w cnn accelerator with many-core processing-in-memory architecture with 896k synapses/mm 2." in *2019 Symposium on VLSI Circuits*. IEEE, 2019, pp. C248–C249.

[7] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[8] "ImageNet Dataset," https://image-net.org/challenges/LSVRC/2012/index.php, 2012 (accessed August 3, 2021).

[9] S. Venkatachalam, E. Adams, H. J. Lee, and S.-B. Ko, "Design and analysis of area and power efficient approximate booth multipliers," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1697–1703, 2019.

[10] D. Yu and L. Deng, *Automatic Speech Recognition.* Springer, 2016.

[11] Y. Su, K. Zhang, J. Wang, and K. Madani, "Environment sound classification using a two-stream cnn based on decision-level fusion," *Sensors*, vol. 19, no. 7, p. 1733, 2019.

[12] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *arXiv preprint arXiv:1905.05055*, 2019.

[13] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixao, F. Mutz *et al.*, "Self-driving cars: A survey," *Expert Systems with Applications*, vol. 165, p. 113816, 2021.

[14] Z. Wu, C. Shen, and A. Van Den Hengel, "Wider or deeper: Revisiting the resnet model for visual recognition," *Pattern Recognition*, vol. 90, pp. 119–133, 2019.

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

[16] "Deep learning basics," https://blog.tensorflow.org/2019/02/mit-deep-learning-basics-introduction-tensorflow.html, accessed: 2022-01-30.

[17] Y. Choi, D. Bae, J. Sim, S. Choi, M. Kim, and L.-S. Kim, "Energy-efficient design of processing element for convolutional neural network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 11, pp. 1332–1336, 2017.

[18] L. Bai, Y. Zhao, and X. Huang, "A cnn accelerator on fpga using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.

[19] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Information Fusion*, vol. 42, pp. 146–157, 2018.

[20] G. Shomron and U. Weiser, "Spatial correlation and value prediction in convolutional neural networks," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 10–13, 2018.

[21] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 701–706.

[22] D. Lee, S. Kang, and K. Choi, "Compend: Computation pruning through early negative detection for relu in a deep neural network accelerator," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 139–148.

[23] K.-C. Chen, M. Ebrahimi, T.-Y. Wang, and Y.-C. Yang, "Noc-based dnn accelerator: a future design paradigm," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, 2019, pp. 1–8.

[24] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[25] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[26] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, vol. 6, 2018.

[27] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 304–317.

[28] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[29] A. Ardakani, C. Condo, and W. J. Gross, "Fast and efficient convolutional accelerator for edge computing," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 138–152, 2019.

[30] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 382–394.

[31] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 749–763.

[32] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient mitchell's approximate log multipliers for convolutional neural networks," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, 2019.

[33] Z. Du, A. Lingamneni, Y. Chen, K. V. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of neural networks for designing highly energy efficient accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1223–1235, 2015.

[34] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, pp. 1–23, 2018.

[35] E. Ozen and A. Orailoglu, "Sanity-check: Boosting the reliability of safety-critical deep neural network applications," in *2019 IEEE 28th Asian Test Symposium (ATS)*. IEEE, 2019, pp. 7–75.

[36] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[37] M. Asadikouhanjani and S.-B. Ko, "Enhancing the utilization of processing elements in spatial deep neural network accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[38] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE, 2018, pp. 1–6.

[39] S. Cass, "Taking ai to the edge: Google's tpu now comes in a maker-friendly package," *IEEE Spectrum*, vol. 56, no. 5, pp. 16–17, 2019.

[40] Q. Sun, T. Chen, J. Miao, and B. Yu, "Power-driven dnn dataflow optimization on fpga," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–7.

[41] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a dataflow and accelerator for sparse deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 711–724.

[42] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[43] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5918–5926.

[44] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[45] A. Delmas, P. Judd, S. Sharify, and A. Moshovos, "Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks," *arXiv preprint arXiv:1706.00504*, 2017.

[46] A. D. Lascorz, S. Sharify, P. Judd, K. Siu, M. Nikolic, and A. Moshovos, "Dpred: Making typical activation values matter in deep learning computing," *CoRR, vol. abs/1804.06732*, 2017.

[47] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need, in 'advances in neural information processing systems (neurips)'," 2017.

[49] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[50] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2014, pp. 27–32.

[51] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan, "Adacomp: Adaptive residual gradient compression for data-parallel distributed training," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[52] M. S. Kim, A. A. Del Barrio Garcia, H. Kim, and N. Bagherzadeh, "The effects of approximate multiplication on convolutional neural networks," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2021.

[53] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, 2018.

[54] Y. Tang, J. Zhang, and N. Verma, "Scaling up in-memory-computing classifiers via boosted feature subsets in banked architectures," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 3, pp. 477–481, 2018.

[55] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira,

C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012.

[56] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[57] S. Zhou, Q. Guo, Z. Du, D. Liu, T. Chen, L. Li, S. Liu, J. Zhou, O. Temam, X. Feng, X. Zhou, and Y. Chen, "Paraml: A polyvalent multicore accelerator for machine learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 9, pp. 1764–1777, 2020.

[58] M. Asadikouhanjani and S.-B. Ko, "A novel architecture for early detection of negative output features in deep neural network accelerators," *accepted to IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[59] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[60] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[61] D. Minoli, K. Sohraby, and B. Occhiogrosso, "Iot considerations, requirements, and architectures for smart buildings—energy optimization and next-generation building management systems," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 269–283, 2017.

[62] M. R. A. Kouhanjani and A. H. Jahangir, "Improving hardware trojan detection using scan chain based ring oscillators," *Microprocessors and Microsystems*, vol. 63, pp. 55–65, 2018.

[63] N. Samimi, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Res-dnn: A residue number system-based dnn accelerator unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 2, pp. 658–671, 2019.

[64] S. Kim, P. Howe, T. Moreau, A. Alaghi, L. Ceze, and V. S. Sathe, "Energy-efficient neural network acceleration in the presence of bit-level memory errors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4285–4298, 2018.

[65] M. Asadikouhanjani and S.-B. Ko, "Enhancing the utilization of processing elements in spatial deep neural network accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[66] A. Ardakani, C. Condo, M. Ahmadi, and W. J. Gross, "An architecture to accelerate convolution in deep neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 4, pp. 1349–1362, 2017.

[67] J. Jo, S. Cha, D. Rho, and I.-C. Park, "Dsip: A scalable inference accelerator for convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 605–618, 2017.

[68] C. Cheng and K. K. Parhi, "Fast 2d convolution algorithms for convolutional neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1678–1691, 2020.

[69] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[70] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, 2016.

[71] G. Shomron and U. Weiser, "Spatial correlation and value prediction in convolutional neural networks," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 10–13, 2018.

[72] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[73] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1251–1258.

[74] H. Zhang and S.-B. Ko, "Design of power efficient posit multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 861–865, 2020.

[75] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[76] C. Cheng and K. K. Parhi, "Fast 2d convolution algorithms for convolutional neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1678–1691, 2020.

[77] M. Asadikouhanjani, H. Zhang, L. Gopalakrishnan, H.-J. Lee, and S.-B. Ko, "A real-time architecture for pruning the effectual computations in deep neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 2030–2041, 2021.

[78] F. Vaverka, R. Hrbacek, and L. Sekanina, "Evolving component library for approximate high level synthesis," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–8.

[79] G. Park, J. Kung, and Y. Lee, "Design and analysis of approximate compressors for balanced error accumulation in mac operator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 7, pp. 2950–2961, 2021.

[80] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[81] X. He, L. Ke, W. Lu, G. Yan, and X. Zhang, "Axtrain: Hardware-oriented neural network training for approximate inference," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.

[82] X. He, W. Lu, G. Yan, and X. Zhang, "Joint design of training and hardware towards efficient and accuracy-scalable neural network inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 810–821, 2018.

[83] S. Venkatachalam and S.-B. Ko, "Design of power and area efficient approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1782–1786, 2017.

[84] D. Esposito, A. G. Strollo, and M. Alioto, "Low-power approximate mac unit," in *2017 13th Conference on Ph. D. Research in Microelectronics and Electronics (PRIME)*. IEEE, 2017, pp. 81–84.

[85] T. Yang, T. Sato, and T. Ukezono, "A low-power approximate multiply-add unit," in *2019 2nd International Symposium on Devices, Circuits and Systems (ISDCS)*. IEEE, 2019, pp. 1–4.

[86] Y. Lu, W. Shan, and J. Xu, "A depthwise separable convolution neural network for small-footprint keyword spotting using approximate mac unit and streaming convolution reuse," in *2019 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2019, pp. 309–312.

[87] M. S. Kim, A. A. Del Barrio, R. Hermida, and N. Bagherzadeh, "Low-power implementation of mitchell's approximate logarithmic multiplication for convolutional neural networks," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 617–622.

[88] M. Ahmadinejad and M. H. Moaiyeri, "Energy-and quality-efficient approximate multipliers for neural network and image processing applications," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[89] "Matlab Speech Recognition Example," https://www.mathworks.com/help/deeplearning/ug/deep-learning-speech-recognition.html, 2017 (accessed August 3, 2021).

[90] "Google Speech Dataset," http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz., 2016 (accessed August 3, 2021).

[91] C. Niemann, M. Rethfeldt, and D. Timmermann, "Approximate multipliers for optimal utilization of fpga resources," in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2021, pp. 23–28.

[92] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, "Area-optimized low-latency approximate multipliers for fpga-based hardware accelerators," in *Proceedings of the 55th annual design automation conference*, 2018, pp. 1–6.

[93] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 145–150.

[94] D. Kim, J. Kung, and S. Mukhopadhyay, "A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 164–178, 2017.

[95] "Environment Sound Dataset," https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/YDEPUT, Updated 2022 (accessed March 26, 2022).

[96] "CIFAR Dataset," https://www.cs.toronto.edu/~kriz/cifar.html, Updated 2022 (accessed March 26, 2022).

[97] F. Vaverka, V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu," in *2020 Design, Automation and Test in Europe Conference (DATE)*, 2020, p. 4.

[98] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[99] "MNIST Dataset," http://yann.lecun.com/exdb/mnist/, Updated 2022 (accessed March 26, 2022).

[100] "Dynamic Function eXchange," https://docs.xilinx.com/v/u/2019.2-English/ug909-vivado-partial-reconfiguration, Updated 2022 (accessed March 26, 2022).

[101] M. A. Rihani, F. Nouvel, J.-C. Prévotet, M. Mroue, J. Lorandel, and Y. Mohanna, "Dynamic and partial reconfiguration power consumption runtime measurements analysis for zynq soc devices," in *2016 International Symposium on Wireless Communication Systems (ISWCS)*, 2016, pp. 592–596.